RL-TR-97-92, Volume I (of two)
Final Technical Report
September 1997

CERTIFICATION FRAMEWORK VALIDATION FOR REUSABLE ASSETS - PROJECT SUMMARY, VOLUME I (OF TWO)

Data & Analysis Center for Software,
KAMAN Sciences Corporation

Sharon Rohde and Karen Dyson,
of Software Productivity Solutions, Inc.

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

Rome Laboratory
Air Force Materiel Command
Rome, New York

This report has been reviewed by the Rome Laboratory Public Affairs Office (PA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

RL-TR-97-92, Volume I (of two) has been reviewed and is approved for publication.

APPROVED: Deborah A. Cerino

DEBORAH A. CERINO
Project Engineer

FOR THE DIRECTOR: John A Graniero

JOHN A. GRANIERO, Chief Scientist
Command, Control, & Communications Directorate

If your address has changed or if you wish to be removed from the Rome Laboratory mailing list, or if the addressee is no longer employed by your organization, please notify RL/C3CB, 525 Brooks Road, Rome, NY 13441-4505. This will assist us in maintaining a current mailing list.

Do not return copies of this report unless contractual obligations or notices on a specific document require that it be returned.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*

2. REPORT DATE
September 1997

3. REPORT TYPE AND DATES COVERED
Final Apr 94 - Feb 97

4. TITLE AND SUBTITLE
CERTIFICATION FRAMEWORK VALIDATION FOR REUSABLE ASSETS - PROJECT SUMMARY, VOLUME I (OF TWO)

5. FUNDING NUMBERS
C - F30602-92-C-0158 T/32
PE - 63728F
PR - 2527
TA - 02
WU - 35

6. AUTHOR(S)
Sharon Rohde and Karen Dyson of Software Productivity Solutions, Inc.

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Data & Analysis Center for Software
KAMAN Sciences Corporation
Griffiss Business & Technology Park
775 Daedalian Drive
Rome, NY 13440-4909

8. PERFORMING ORGANIZATION REPORT NUMBER
N/A

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Rome Laboratory/C3CB
525 Brooks Road
Rome, NY 13441-4505

10. SPONSORING/MONITORING AGENCY REPORT NUMBER
RL-TR-97-92, Vol I (of two)

11. SUPPLEMENTARY NOTES
Rome Laboratory Project Engineer: Deborah A. Cerino/C3CB/(31) 330-2054

12a. DISTRIBUTION AVAILABILITY STATEMENT
Approved for public release; distribution unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*
The purpose of this effort was to further develop, apply, and validate the Rome Laboratory Software Certification Framework for designating various levels of confidence in the quality of reusable software. This effort fine-tuned the Framework's ability to distinguish between reusable assets of differing quality.

The effort resulted in a two volume final technical report. Volume I - the Project summary, describes the complete contractual effort. The report discusses how the quality assessment methodology, techniques, and metrics embodied within the Rome Laboratory Software Quality Framework (SQF) could be applicable to the certification of reusable assets. The report discusses potential upgrades and re-engineering the Rome Laboratory Software Quality Framework (SQF). In addition, it also overviews the application of the Certification Framework to a small set of software components (i.e., source code). Volume II - Certification Field Trial, fully details the procedures, collection forms, results, and lessons learned from the application of the certification process to the software components.

14. SUBJECT TERMS
Software Certification, Software Assessment and Evaluation

15. NUMBER OF PAGES
280

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT
UNCLASSIFIED

18. SECURITY CLASSIFICATION OF THIS PAGE
UNCLASSIFIED

19. SECURITY CLASSIFICATION OF ABSTRACT
UNCLASSIFIED

20. LIMITATION OF ABSTRACT
UL

Table of Contents

List of Figures

List of Tables

Contributors to the ATD Project

Listed in alphabetical order, the following persons contributed to the ATD Project:

Lynda L. Burns, Software Productivity Solutions

Deborah A. Cerino Rome Laboratory of the U.S. Air Force Materiel Command

Karen A. Dyson, Software Productivity Solutions, Inc.

Jeffrey A. Heimberger, Software Productivity Solutions, Inc.

Beth Layman, Lockheed Martin Corporation

Holly G. Mills, Software Productivity Solutions, Inc.

Annette Myjak, Software Productivity Solutions, Inc.

Sharon L. Rohde, Software Productivity Solutions, Inc.

Tom Strelich, GRC International

Steven Wee, Software Productivity Solutions, Inc.

Executive Summary

Reuse certification is a technology that shows great promise in providing the assurance of quality of reusable assets that is essential to achieving greater levels of reuse. The Certification Framework developed under Rome Laboratory's Certification of Reusable Components (CRC) effort developed the first comprehensive multi-domain approach to developing certification processes that are effective at assuring quality. This advanced technology demonstration effort, Certification Framework Validation for Reusable Assets, was undertaken partly in parallel with the CRC effort to broaden the applicability of the reuse certification technology to the commercial domain.

Certification technology is of interest to a broader audience than the reuse context implies because it deals with the same techniques and tools that are used in mainstream software development. Thus the results of this effort are equally applicable to those who specialize in reuse (e.g., reuse repositories, reusers of software, or developers of reusable assets) as well as to software developers and maintainers. Researchers involved in developing or improving verification techniques, or in software process improvement, will be interested in the field trials, defect model, and cost/benefit models.

This effort has successfully demonstrated reuse certification of a C++ asset in the second certification field trial. Both certification field trials provided numerous valuable lessons learned through practical application of the products of the CRC and of this effort. Perhaps the most valuable lesson learned was that the four certification techniques (compilation, automated static analysis, code inspection, and testing) were much more effective *in combination* than any one technique by itself because they tended to find different types of defects.

As a result of this effort, many useful work products are now available for technology transfer. These work products include the following:

- Ada and C++ Code Inspection Checklists
- Guidelines for Building in Portability and Correctness
- Portability and Correctness Indicators (metrics)
- Field Trials Lessons Learned
- CRC Web Pages & Interactive Demonstration

These work products have advanced the state of the art of reuse and certification technologies.

1.0 Introduction

The U. S. Air Force Rome Laboratory (RL) reuse certification initiative was an outgrowth of RL's rich 20-year legacy of research into software reliability and quality software. The foundation of the reuse certification initiative is the RL-sponsored studies that developed a Certification Framework for reusable software components.

This effort, an Advanced Technology Demonstration (ATD) called Certification Framework Validation for Reusable Assets, was a continuation and augmentation of the work performed by Software Productivity Solutions, Inc. (SPS) for RL in the reuse certification area. The work is related to two contracts:

- Certification of Reusable Software Components (CRC), Contract No. F30602-94-0024.
- Reuse-Based Software Quality Framework for Certification (RC-SQF), Contract No. F30602-92-C-0158.

The objective of the CRC Program was to develop a practical, cost-effective approach for certification of reusable components in order to stimulate software reuse throughout industry and encourage the emergence of a commercial components market. A prime development objective for the Certification Framework is that it include only the practical, usable, and cost-effective subset of reliability techniques that improve confidence in reusable software.

The RC-SQF contract developed a fully automated software metrics tool for evaluation of the quality of reusable Ada software components. The RC-SQF breaks down the quality scores into software attributes called criteria. Low scores for a particular criterion indicate the need for further certification activities. The certification guidance portion of the RC-SQF Final Technical Report assists the certifier by translating low criteria scores into needed certification activities such as specific types of testing. The RC-SQF quality evaluation framework is a subset of the Rome Laboratory Software Quality Framework (RLSQF) tailored for automatability from Ada code.

1.1 Objectives and Tasks

This effort had two main objectives. The first main objective was to upgrade and expand the application of the RL Software Quality Framework (SQF). Under this effort, we identified needed framework upgrades, developed a re-engineering approach, and extracted useful techniques from the SQF applicable to certification. The second main objective was to further develop, apply and validate the RL Certification Framework (CF) initially developed under the CRC contract. Under this effort, we expanded the RL Certification Framework to apply to multiple domains and to an additional pilot site which is a commercial reuse organization.

The following set of tasks are aimed at the first objective described above. These tasks and their results are described in following sections as noted beside the task. The first group of tasks in section 2 deal with SQF upgrades and modifications. The second set of tasks in Section 3 deal with extracting valuable techniques from the SQF to apply to reuse certification.

Task	Section
SQF Recommended Modifications	2.3
SQF Re-engineering Approach	2.4
Re-engineered SQF Results	2.5
Re-engineered Tailoring Approach	2.5.4
Re-engineered SQF Example: Portability	2.5.5
Support of PSM PE Working Group	2.6
Analysis of SQF for Certification	3.1
Code Inspection Checklists	3.2
Ada Style Guideline Checks	3.3
Guidance for Building in Correctness	3.4
Predictive Model	3.5

The following set of tasks are aimed at the second objective described above. These tasks and their results are described in following sections as noted beside the task.

Task	Section
Field Trial with C++ Code Asset	4.1.1
Support Pilot Sites	4.1.2
Develop Commercial Standards - RIG	4.2.1
Survey of Commercial Organizations	4.2.2
CRC Web Pages	4.3

1.2 Summary of Accomplishments

Numerous work products resulted from this effort, illustrated in Figure 1-1. All of these work products are documented in this FTR:

- Re-engineered SQF Design incorporating the quality blueprint concept
- Portability Guidelines and Indicators
- Correctness Guidelines and Indicators

- Ada Code Inspection Checklist
- C++ Code Inspection Checklist
- Reuse in Commercial Organizations Survey Results
- Second Certification Field Trial Process & Results
- Automated Ada Style Guideline Checks in AdaQuest static analysis tool
- Analysis of BLSM-I defect data
- State-of-the-Art Report on Reuse Library
- CRC Web Pages and Demonstration

The ATD Project supported the development of the IEEE standard 1420.1a Asset Certification Framework, and can be ordered from that organization.

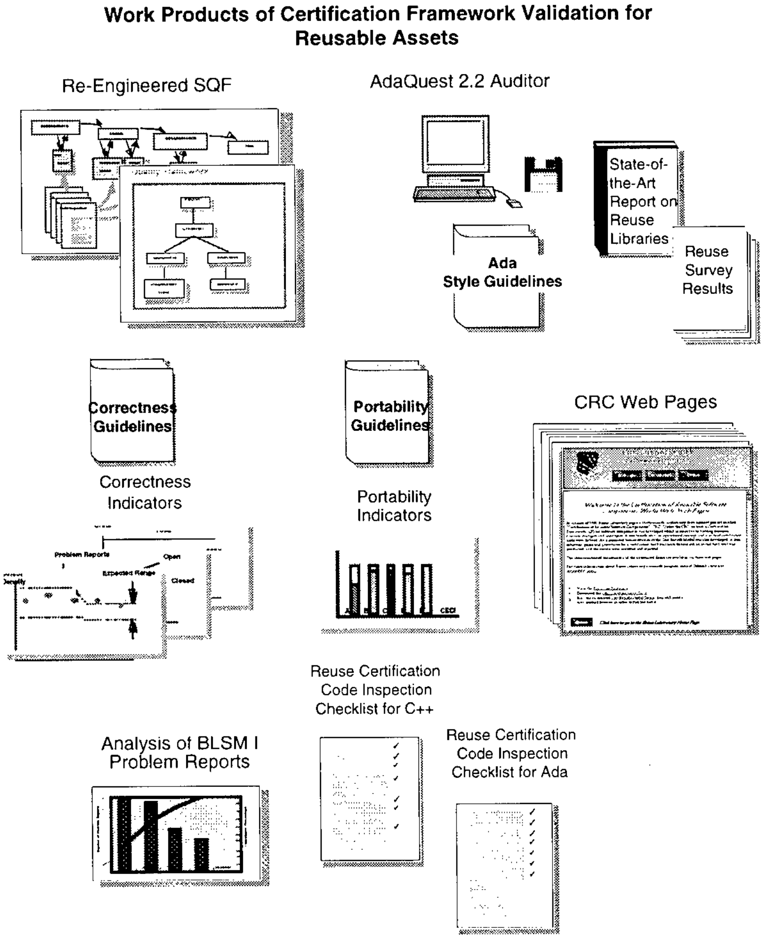

Work Products of Certification Framework Validation for Reusable Assets
Re-Engineered SQF
AdaQuest 2.2 Auditor
State-of-the-Art Report on Reuse Libraries
Ada Style Guidelines
Reuse Survey Results
Correctness Guidelines
Portability Guidelines
CRC Web Pages
Correctness Indicators
Problem Reports
Open
Expected Range
Closed
Portability Indicators
Reuse Certification Code Inspection Checklist for C++
Analysis of BLSM I Problem Reports
Reuse Certification Code Inspection Checklist for Ada

Figure 1-1. Work Products Resulting from this Effort

2.0 Software Quality Framework Upgrade

This section documents the results of the set of tasks that deal with upgrades to the RL Software Quality Framework (SQF). This section begins with a short overview of the previous work on RC-SQF effort as a lead in to this effort in section 2.1. Next, in section 2.2, is an overview of the RL SQF to provide background information for the subsequent sections. Included in this section are newly developed graphical representations of the framework structure as well as analysis of the framework's strengths.

Section 2.3 provides the results of analysis of recommended modifications to the SQF which led to the re-engineering concept discussed in section 2.4. Not all of the recommended modifications were implemented as part of this effort. Instead, a portion of the framework was re-engineered to provide an example. The selected portion is a top-to-bottom thread through the framework for the factor of Portability, and the results are documented in section 2.5.

The SQF re-engineering work was influenced by participation in the Joint Logistics Commanders Practical Software Measurement (PSM) working group for software product engineering. The group is in the process of creating a measurement framework of which software quality is a major part. Our participation is documented in section 2.6.

2.1 Evolution of the RC-SQF

The RC-SQF effort was the initial attempt to derive techniques useful for reuse certification from the RL Software Quality Framework. The RC-SQF contract developed a fully automated software metrics tool for evaluation of the quality of reusable Ada software components. The reuse certifier can use this tool to identify weaknesses of a component by examining the quality scores. The RC-SQF breaks down the quality scores into software attributes called criteria. Low scores for a particular criterion indicate the need for further certification activities. The certification guidance portion of the RC-SQF Final Technical Report [SPS94F] assists the certifier by translating low criteria scores into needed certification activities such as specific types of testing.

The RC-SQF quality evaluation framework is a subset of the Rome Laboratory Software Quality Framework (SQF) tailored for automatability from Ada code. The system is an integration of RL's Quality Evaluation System (QUES) tool and General Research Corporation's commercial tool, AdaQuest™. AdaQuest is the first commercial tool to interface with RL's QUES. AdaQuest collects the software metrics from Ada code, and QUES computes and reports on the quality scores.

Over the course of this effort, in conjunction with the development of a certification process for code assets on the CRC contract, this RC-SQF evolved into a part of the certification toolset. The AdaQuest tool, with added capabilities to check for violations of Ada style guidelines (as described in section 3.3), is used in the Static Analysis step of the certification process.

2.2 Software Quality Framework (SQF) Overview

This section contains an overview of the RL SQF, to provide background information as a basis for understanding the modifications to the SQF proposed in this report. The SQF was developed by RL to identify key software quality issues and to provide a valid methodology for specifying software quality requirements and for measuring achieved quality levels of software products released incrementally during the software life cycle.

The SQF is the culmination of research beginning in 1976, and the framework continues to evolve to keep pace with the state of the art of software engineering and the ever-increasing demands of the DoD customers. The trend has been for the framework to grow to encompass more quality concerns and measures.

The SQF is intended to satisfy three needs relative to software quality:

- to thoroughly specify the level of quality required
- to predict the quality of the end product early in the software development life cycle
- to measure the achieved quality levels.

The SQF is a collection of techniques used throughout the software development life cycle. It is mainly a series of product inspections, as illustrated in Figure 2-1, but also includes techniques of traceability analysis, target computer resource utilization (CRU) measurement, problem report tracking, and complexity metrics.

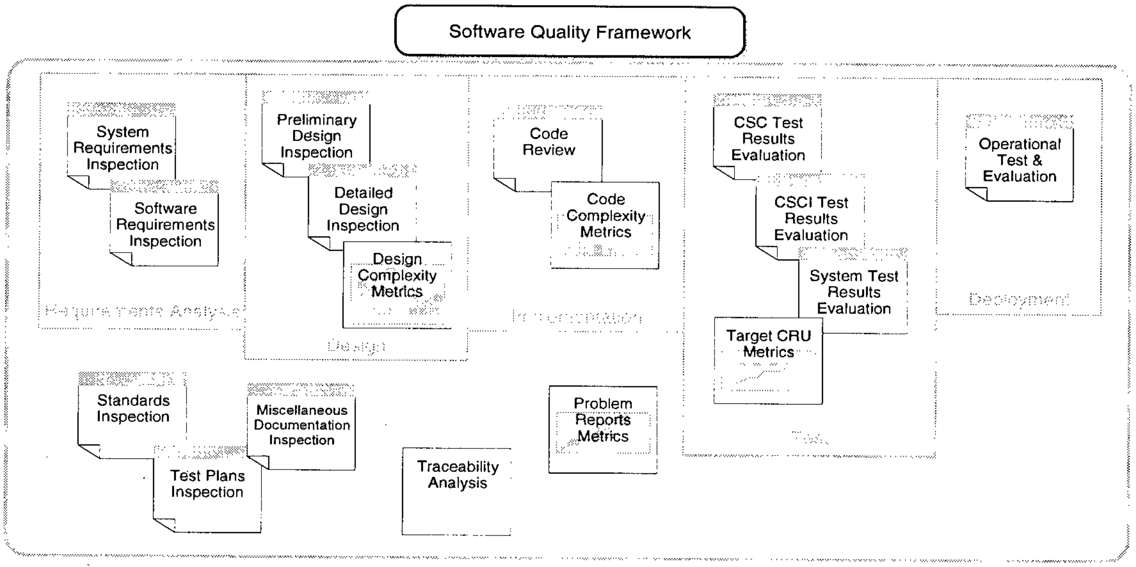

Software Quality Framework
System Requirements Inspection
Software Requirements Inspection
Requirements Analysis
Preliminary Design Inspection
Detailed Design Inspection
Design Complexity Metrics
Design
Code Review
Code Complexity Metrics
Implementation
CSC Test Results Evaluation
CSCI Test Results Evaluation
System Test Results Evaluation
Target CRU Metrics
Test
Operational Test & Evaluation
Deployment
Standards Inspection
Test Plans Inspection
Miscellaneous Documentation Inspection
Traceability Analysis
Problem Reports Metrics

Figure 2-1. SQF is a Collection of Techniques.

The foundation of the SQF is the software quality model, shown in Figure 2-2, that establishes a hierarchical relationship between a user-oriented quality factor at the top level and software-oriented attributes at the second and third levels (criteria and metrics, respectively). Software quality is measured and predicted by the presence, absence, or degree of identifiable software characteristics.

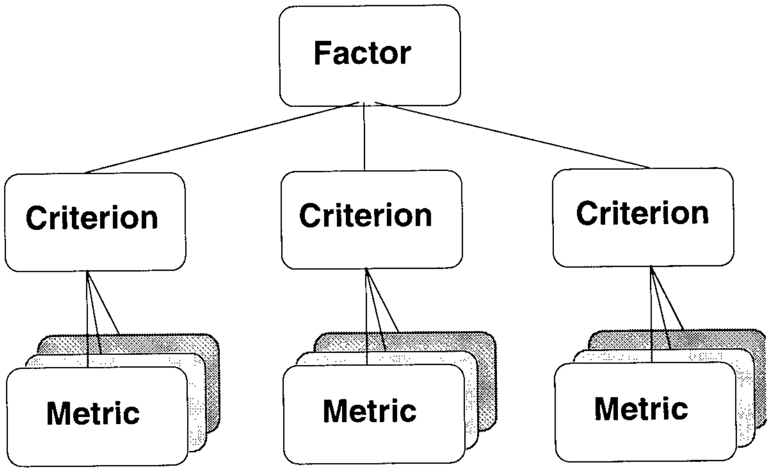


Figure 2-2. The SQF Quality Model Hierarchy.

The hierarchical quality model is perhaps the most well-known and accepted aspect of the SQF. At the top level of the quality model, the 13 quality factors define a quality vocabulary that allow one to be specific when identifying quality concerns. The hierarchical nature of the model partitions quality concerns into convenient packages that can be tailored to suit the needs of each program. Prioritizing and ranking concerns by these SQF factors focuses attention on the issues of importance and helps in making trade-offs.

Figure 2-3 illustrates the relationship between the 13 quality factors, and the 29 criteria found in the latest version of the framework [SQF95]. As this diagram reveals, the factor/criterion relationships are somewhat complex. Not all quality factors are independent. Only three, Usability, Efficiency, and Integrity are completely independent of other factors. Many factors are closely related and rely on common criteria such as Modularity, Simplicity, Self-Descriptiveness, Generality, and Independence.

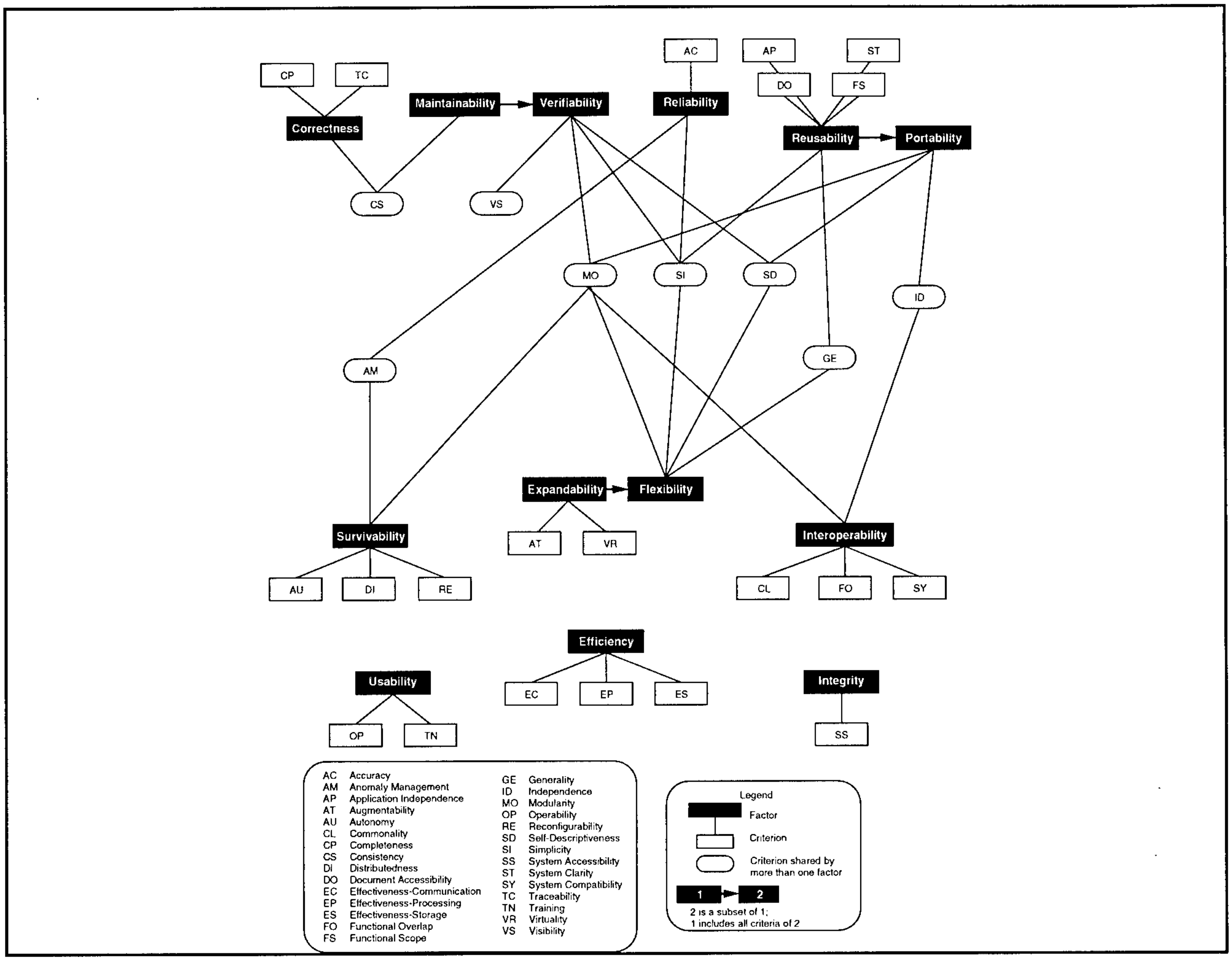


Figure 2-3. The SQF Factor/Criteria Model.

2.2.1 Structure of the SQF

In addition to the hierarchical quality model, the SQF is also subdivided into data collection forms for each of nine phases of the software life cycle shown in the table below. Quality is evaluated separately in each phase, thus the framework is actually composed of 9 independent mini-frameworks.

All phases use the same factor/criteria model, but what differs is the lesser or greater extent of coverage of the factors and criteria in each phase. For example, in phases D, E, F, and I, not all 13 factors are available to measure. Phase B has the most complete coverage and phases F and I have the sparsest coverage of criteria and metrics.

Table 2-1. The RLSQF Data Collection Forms.

Data Collection Form	Life Cycle Phase
A	System Requirements Analysis/Design
B	Software Requirements Analysis
C	Preliminary Design
D	Detailed Design
E	Coding and CSU Testing
F	CSC Integration and Test
G	CSCI Testing
H	System Testing
I	Operational Test and Evaluation

It is difficult to draw a diagram of the SQF structure in its entirety. If we attempt to show the factors, criteria, and metrics for all phases in a single diagram, we get "spaghetti" (see Figure 2-4). This difficulty is an indicator of the size and complexity of the framework.

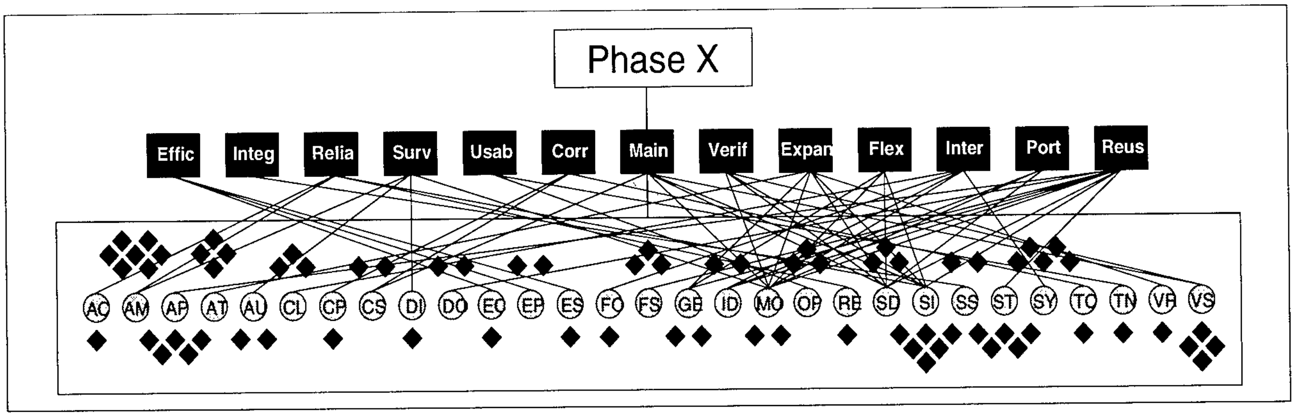


Figure 2-4. Attempting to Diagram the SQF Structure.

Instead, we show the criteria and metrics with the next level of detail, the metric elements, which are the measures that make up metrics. The diagram in Figure 2-5 is simpler because metrics belong to a single criterion. In fact, the metric identifier is simply the criterion abbreviation with a numerical suffix. This type of diagram is useful for comparing the number of metric elements among the criteria and metrics, and for illustrating automation of the metric elements.

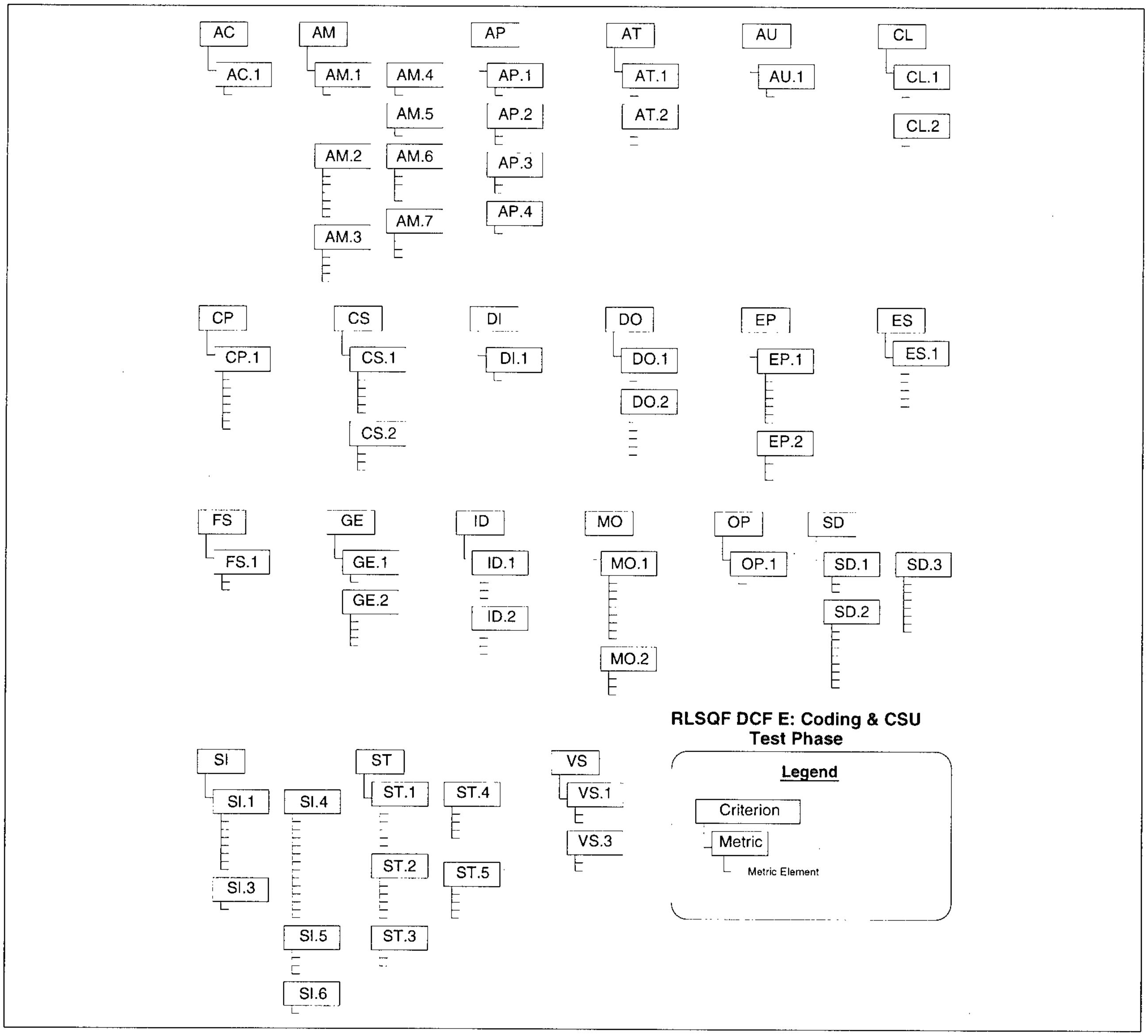


Figure 2-5. Metric Elements for a Single Phase of the SQF.

2.2.2 Metric Elements

Metric elements are either Yes/No (Boolean) questions, or are ratios of numerical questions designed to evaluate to a value between zero and one. Questions are organized into Data Collection Forms (DCFs), one for each phase of the software development life cycle. Figure 2-6 contains an excerpt from the Phase C (Preliminary Design) DCF. Each question has a unique identifier that includes the two-letter criterion abbreviation, the metric number, the question number, and the phase suffix. Each question is associated with one level of the static structure hierarchy (System, CSCI, CSC, or CSU). As Figure 2-6 shows, different criteria and static structure levels are combined within a DCF.

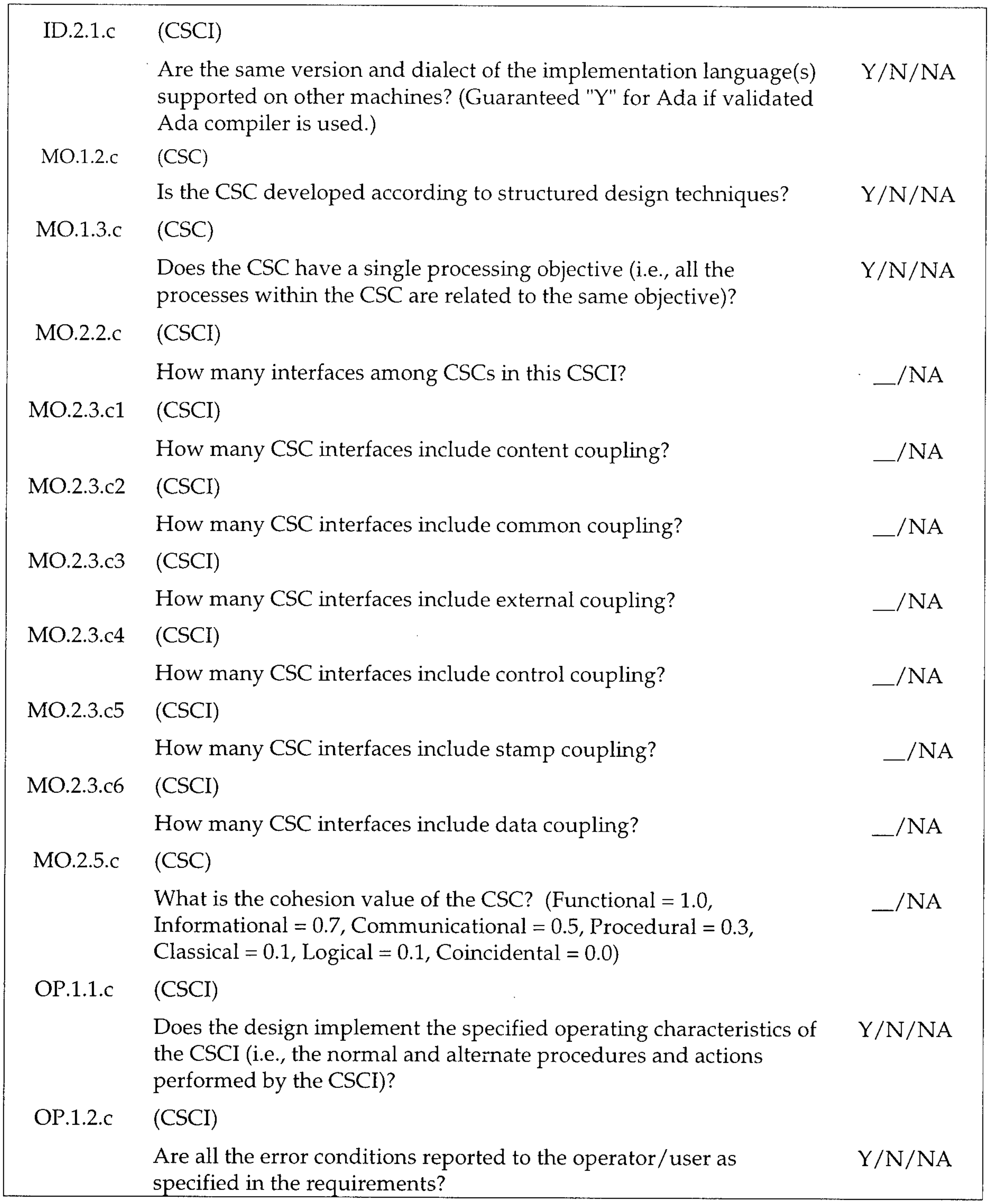
Every question may have an answer (which is Yes or No, or a value depending on the type of question), or it could be Not Applicable (NA). When computing quality scores, NA answers do not affect the score either positively or negatively. If any question in a metric element ratio is NA, the entire metric element is NA.

Metric elements are computed in the metric scoring equations. Table 2-2 below shows the metric scoring equations for the questions shown in Figure 2-6. Each metric element is enclosed in a weight function, designated by "1.0( )". By default, each metric element is equally weighted, thus the weight is "1.0". The SQF scoring methodology allows users to adjust weights.

The "W.AVG( )" function computes the weighted average of the operand for all subordinate (i.e., child) units. The equation for MO.2, for example, shows that the CSCI score is augmented by the weighted average of the MO.2 score of all of its subordinate CSCs.

Table 2-2. Metric Scoring Equations for the Questions in Figure 2-6.

Metric	Level	Scoring Equation
ID.2	CSCI	ID.2.1.c
MO.1	CSCI	W.AVG(MO.1)
	CSC	1.0(MO.1.2.c) + 1.0(MO.1.3.c)
MO.2	CSCI	1.0(1.0 - (MO.2.3.c1 + MO.2.3.c2 + MO.2.3.c3) / MO.2.2.c) + 1.0(1.0 - (MO.2.3.c4 + MO.2.3.c5) / (2 * MO.2.2.c)) + 1.0(MO.2.3.c6 / MO.2.2.c) + 1.0(W.AVG(MO.2))
	CSC	MO.2.5.c
OP.1	CSCI	1.0(OP.1.1.c) + 1.0(OP.1.2.c) + ...



ID.2.1.c	(CSCI)	
	Are the same version and dialect of the implementation language(s) supported on other machines? (Guaranteed "Y" for Ada if validated Ada compiler is used.)	Y/N/NA
MO.1.2.c	(CSC)	
	Is the CSC developed according to structured design techniques?	Y/N/NA
MO.1.3.c	(CSC)	
	Does the CSC have a single processing objective (i.e., all the processes within the CSC are related to the same objective)?	Y/N/NA
MO.2.2.c	(CSCI)	
	How many interfaces among CSCs in this CSCI?	__/NA
MO.2.3.c1	(CSCI)	
	How many CSC interfaces include content coupling?	__/NA
MO.2.3.c2	(CSCI)	
	How many CSC interfaces include common coupling?	__/NA
MO.2.3.c3	(CSCI)	
	How many CSC interfaces include external coupling?	__/NA
MO.2.3.c4	(CSCI)	
	How many CSC interfaces include control coupling?	__/NA
MO.2.3.c5	(CSCI)	
	How many CSC interfaces include stamp coupling?	__/NA
MO.2.3.c6	(CSCI)	
	How many CSC interfaces include data coupling?	__/NA
MO.2.5.c	(CSC)	
	What is the cohesion value of the CSC? (Functional = 1.0, Informational = 0.7, Communicational = 0.5, Procedural = 0.3, Classical = 0.1, Logical = 0.1, Coincidental = 0.0)	__/NA
OP.1.1.c	(CSCI)	
	Does the design implement the specified operating characteristics of the CSCI (i.e., the normal and alternate procedures and actions performed by the CSCI)?	Y/N/NA
OP.1.2.c	(CSCI)	
	Are all the error conditions reported to the operator/user as specified in the requirements?	Y/N/NA

Figure 2-6. Excerpt from Phase C (Preliminary Design) Data Collection Form.

2.2.3 Important SQF Concepts

The following paragraphs summarize important concepts embodied in the SQF approach.

- **Many facets of quality**. The SQF view of quality is many-faceted, thus the 13 quality factors, which each represent different aspects of quality. Similarly, the CRC approach to reuse certification has identified three aspects of quality that are of universal concern: Correctness, Understandability, and Completeness. Specific reuse applications may have additional concerns such as Portability, Efficiency, Safety, etc.
- **Importance of a user/customer-oriented point of view.** SQF factors reflect the user's or customer's view of quality.
- **Ranking factors is valuable.** Customers can rank the factors to indicate the relative importance of the aspects of quality. This ranking serves as guidance when the inevitable trade-offs between quality and resources must be made.
- **Translation of abstractions.** The framework translates somewhat abstract or intangible user-oriented "factors" into concrete software characteristics that software engineers can relate to.
- **Full life cycle approach.** SQF is a full life cycle approach. The framework contains a quality model to predict the quality of the end product from early life cycle artifacts.
- **Product inspection methodology.** SQF is a collection of techniques (see Figure 2-1), but is basically a set of product inspections.
- **Blueprint for quality.** The framework can be viewed as a set of guidelines or principles to design quality into the product--a "blueprint for quality".

2.3 Recommended Modifications to SQF

The objective of this task was to identify modifications to the SQF in order to improve its usability and applicability. Sources of recommendations include interviews with experts knowledgeable in the SQF, interviews and informal discussions with SQF users in the RL SQT2 Consortium (described below), documented recommendations gathered on the Quality Evaluation System (QUES) contract F30602-88-C-0019 as well as analysis by software measurement experts on SPS' staff. Sections 2.3.1 through 2.3.7 describe the recommended modifications.

Overview of the RL Software Quality Technology Transfer (SQT2) Consortium

The Consortium, sponsored by RL, was launched in August 1991, and its primary objective was to transfer RL's software quality technology to industry. Participating organizations entered into Cooperative Research and Development Agreements (CRADAs) with RL. RL provided training in the technology, tools (including Quality Evaluation System (QUES)), and consulting and data analysis services. Members conducted pilot application projects, applying the SQF methodology to in-house software development programs. Members provided their data to RL so that RL could perform multi-project evaluation of the SQF. Members also reported to RL on the costs and benefits of the technology.

Members included Northrop Grumman Melbourne, Hughes Aircraft of Canada, CTA SoHaR, Frontier, and Kaman Sciences. Members agreed to collect SQF data for the factors Reliability and Maintainability so that there would be common data across all pilot application projects. The long-term goal was to validate the SQF scoring approach by comparing the SQF scores to independent measures of Reliability and Maintainability. RL retained the project data in a Consortium repository. Four participating organizations submitted data to RL, and two have ongoing projects.

Evolution of the SQF

There is a continuous need to keep the SQF framework up to date with the state of the art in software engineering. The most recent formal review of the framework (prior to this effort) occurred in the late 1980's, as part of the Quality Evaluation System contract. The results of this review were documented in [SAI89]. The varied sources of recommendations included analysis of related research and development efforts, surveys, and working group meetings. Approved recommendations were incorporated in the 1991 version of the SQF [DYS91A] that was implemented in the Quality Evaluation System (QUES) tool, but many were tabled pending future research and analysis. Recommendations for change in this technical report are equally applicable to the 1991 SQF baseline, as well as to the most recent Version 1.5 [SQF95].

The detailed nature of the questions in the SQF data collection forms necessitates inherent assumptions about standards, methods, and implementation languages. Because these characteristics are subject to change, it is important to document what these assumptions are to make it easier to maintain the framework. In 1989, for example, the recommendations from many sources were to incorporate the DoD-STD-2167A standard nomenclature, and this was accomplished in the 1991 version. Now the recommendations are to remove the 2167A dependencies!

Categorization of Changes

The recommended changes to SQF can be categorized as one of the following:

- structure and/or content
- methodology of application
- conformance to standards
- scoring
- automation
- training/packaging
- management of the framework.

Each of the above types of changes is discussed in the following subsections.

2.3.1 Structural and Content Changes

Most of the recommended modifications fall into the category of structural and content changes because this is the broadest category. When discussing changes to the SQF structure, it is important to realize that there are three dimensions to the organizational structure: the factor/criterion/metric hierarchy, the architecture (static structure) hierarchy, and the data collection form breakdown.

The nature of the recommendations in this category range from high level concerns such as the makeup of the factors and reorganization of the data collection forms, to details such as the wording of specific questions.

Although there were no recommendations to alter the basic factor, criterion, and metric hierarchy and nomenclature, there were proposed changes such as the following:

- Add or delete factors
- Delete criteria or reallocate criteria among factors
- Add new metrics or metric elements

The argument for deleting certain factors (such as Efficiency, Integrity, Expandability, and Interoperability) is that they should be specified concretely as functional, performance, or interface requirements, rather than as quality factors. While quality factors alone may not be adequate from an acquisition point of view, this argument neglects the need for predictive indicators for these factors early in the life cycle before the specified requirements can be adequately tested.

New factors have been proposed, such as Trustworthiness and Safety, to expand the coverage of the framework.

Need for Tailoring Support

Tailoring by subsetting is a technique that allows users to cut the framework down to size to reduce the cost of data collection. Users might like to evaluate all 13 quality factors, but typically cannot afford the necessary data collection. Therefore, users select the factors that are most important to their application and tailor the data collection to just those factors.

Another aspect of tailoring is to be able to prune away the portions of the framework that are not applicable, for example, because of the nature of the system's requirements, the methodology, or the implementation language. This type of pruning is not easy to do because the threads run all throughout the framework and are difficult to trace. Questions must be evaluated one by one, and this is time consuming. The tailoring process could be made simpler and more efficient by redesigning the framework for tailoring.

Removing Ambiguity

Consortium (and other) users have complained that many questions are either subjective, or are not stated unequivocally. Too much effort is needed to interpret these questions. Sometimes the operative word in the question is unknown, or could have multiple interpretations. For example, the Completeness question:

```
CP.1.2.e     How many data items are identified?
```

To answer this question, we need to know what constitutes "identification." One interpretation is that CP.1.2.e is really asking for the total number of data items referenced in the unit. The word "identified" in this case simply means that a reference is made to a data item in the unit, not that the data item is declared within the unit. Therefore we can count data items that are declared outside the unit. On the other hand, identification could mean that documentation about the data item exists, such as descriptive comments. It is impossible, however, to confidently interpret this question without knowing how it will be used in the metric element equation, and being able to interpret the other related questions that constitute its metric element.

In some cases, we can simply restate the question or define certain terms in a glossary to remove ambiguity. Frequently, interpretation is improved by making the definitions implementation language-dependent. The 1991 SQF addressed some of these needs by adding Ada-specific examples, such as:

AM.4.1.e Is recovery made (e.g., exception handlers or other means) from all detected hardware faults (e.g., arithmetic faults, hardware failure, clock interrupt)?

ST.2.1.e How many unique execution paths are in the CSU, including those caused by a RAISE statement?

There is a conflict between the desire to make the framework methodology- and language-independent, and the need to have the questions be specific enough to be answered consistently and potentially to be automated. One possible approach is to state the underlying principle or guideline in as language-independent a fashion as possible, and then to provide language-specific examples. More details about this "guidelines" approach are provided in Section 2.4.

2.3.2 Methodology of Application

Methodology related recommendations deal with how the framework is used within a software development project. This encompasses many aspects such as sources of data, how often data is collected, reported, and analyzed, and how to handle non-compliance, and how to tailor the framework. One difficulty that users have experienced is in adapting the framework to their unique software development process and to their organization.

There is currently a lack of guidance about how to make use of the information that the SQF provides. One also has to be wary of facile conclusions when interpreting answers to the SQF questions. When a problem is uncovered, the appropriate corrective action may not be to simply convert a No into a Yes. In many cases the metrics are most appropriately used as indicators rather than absolute measures. If Accuracy is a concern, for example, the framework highlights the need for an error budget, early in the life cycle, that allocates errors among the various components of the system. If there is no such error budget, then that is a severe impact to Accuracy, and the appropriate action is to generate the error budget. But the lack of the error budget is probably indicative of a general lack of adequate error analysis that may have far-reaching implications in the design of the system. In the case of Simplicity measures, for example, the framework may indicate that a unit has above average complexity. The appropriate action in response to this finding is probably not to send the unit back for rework, but instead to target it for additional scrutiny such as a code walkthrough, or more comprehensive test coverage.

2.3.3 Conformance to Standards

The purpose of these recommendations are to bring the framework up to date with current standards and practices.

- Remove DoD-STD-2167A dependencies
- Incorporate MIL-STD-498 or relevant IEEE or ISO standards
- Incorporate commonly approved practices and de facto standards such as the SPC Ada Quality and Style Guidelines.

One possible approach to removing 2167A dependencies is to reorganize the DCFs by product and/or activity, rather than by the 2167A life cycle phases. Each question would then be associated with a specific product or activity.

2.3.4 Scoring

Many of the suggestions evaluated by the QUES contract working group were tabled with recommendations for further study, and modifications to scoring is one such suggestion. Because the scoring equations are still theoretical, there is a need for empirical evidence that the computed quality factor scores are correlated to the quality of the final product. It does not make sense to alter the scoring methods (such as considering alternative factor/criteria models, relative weighting schemes, or modifying metric elements) based on what amounts to intuition and conjecture.

The RL SQT2 Consortium is an effort to provide needed empirical data, but efforts are hampered by the scope of the framework. The Consortium was forced to limit itself to consideration of just 2 of the 13 quality factors, Reliability and Maintainability. Another framework characteristic that constrains empirical validation is the amount of time that is required to amass the data; some correlating measures may not be available until after deployment of the system.

Aspects of scoring that have an impact on the usability of the framework are promising areas for improvement despite the lack of validation. The way that the quality evaluation information is *presented* is an important consideration, and we have seen the need for both high-level summarization of quality suitable for the customer or program manager, as well as low-level detailed results for the developers. System- or CSCI-level factor scores might be something to present at major reviews, but they are not necessarily helpful in identifying a problem at a level of detail where it can be fixed. When using the framework as an inspection checklist, for example, the pertinent information may be the units that are found to be non-compliant.

Complementary relationships between factors are adequately modeled by the factors having criteria in common. One aspect that the scoring in the current framework does not support, however, is the concept of *inverse relationships* among factors. For example, there may be a trade-off between Maintainability and Efficiency. The inverse relationships among factors are discussed in [BOW85] but are not supported by the scoring of the framework. One suggestion is to use the same metric elements for both factors, scoring positively for one and negatively for the other.

2.3.5 Automation

The cost of application of the SQF methodology is a major impediment to its acceptance. The most expensive aspect is data collection because most questions must be answered by a knowledgeable person. Automation of data collection is seen as the best solution for reducing the cost to apply, and users are constantly clamoring for more automation. When limited in resources to apply the SQF, users have a tendency to apply just the automated portion of the framework (exemplified by the RC-SQF framework).

Consortium users have requested expanding automation to cover more of the framework, providing automated tools for implementation languages (in addition to Ada) such as C and C++, and improving the automation for FORTRAN.

2.3.6 Training and Packaging

One of the hurdles that new users of the SQF methodology must overcome is understanding how to use it, and how to apply it in particular to their unique situation. The size and complexity of the framework makes this initial task daunting, and many potential users are overwhelmed before ever getting to the application stage.

The need for improved training in the methodology has been identified, as well as the need for "packaging" the technology to simplify it and to increase its acceptance. However, no specific recommendations have yet been received to identify solutions for these needs.

2.3.7 Management of the Framework

Management-related recommendations are concerned with the custody of, dissemination of, and future directions of the framework. One suggestion is to apply configuration management to the framework, via a Configuration Control Board (CCB) to establish baselines and to control changes.

Traceability

Making each question in the framework traceable to a standard or reference would have several advantages:

- reduce interpretation difficulties by providing a backup source for more information
- remove objections based on the argument that the framework is not "validated"

- simplify future updates as standards or practices change

The reference tag idea is illustrated in the section on the Guidelines Approach which follows.

2.4 SQF Re-engineering: The Guidelines Approach

The underlying guidelines that are the SQF blueprint for building quality into software can be extracted by examining the questions in the DCFs and "reverse-engineering" the framework. The approach embodied in the SQF is that the guidelines should appear as explicit requirements, and the requirements should be traced down through design, code, and testing. Inspections should be made at each stage of the life cycle to ensure that the requirements are being implemented, and finally the requirements should be explicitly tested at the system level.

The guidelines can be thought of as forming underlying foundation of the framework, a structure upon which both inspections and product measurement are built. Each guideline would be associated with one or more products or activities, and would have a reference. The guideline would be stated as a statement rather than a question, and ideally in a methodology- and language-independent manner. Each metric element might have dependencies, such as implementation language or development methodology, which should be explicitly documented. Explicit dependency identification is very helpful when tailoring the framework.

2.4.1 Motivation for Reengineering and Repackaging the SQF

In attempting to develop a broader and higher-level perspective for both the certification effort and the SQF, we have concluded that both fall into the category of quality assessment for product engineering. We believe that both certification and SQF can make significant contributions to the effort sponsored by the Joint Logistics Commanders to develop "Practical Quality Measurement" guidance for product engineering (Our participation in the technical working group for this effort is discussed in Section 2.6) However, in order to participate in a working group of this type, we must be able to communicate the value of SQF. The key concepts described in subsection 2.2.3 are the core ideas embodied in the SQF approach that we have already extracted by our analysis to date. In addition, we feel there is a need to repackage the SQF to accomplish the following:

- elucidate the underlying software engineering guidelines, or principles, from which the data collection forms derive
- separate out the various techniques, such as distinguishing traceability analysis and problem report metrics from the inspection checklists
- describe lessons learned about the predictive model

These concepts are illustrated in Figure 2-7.

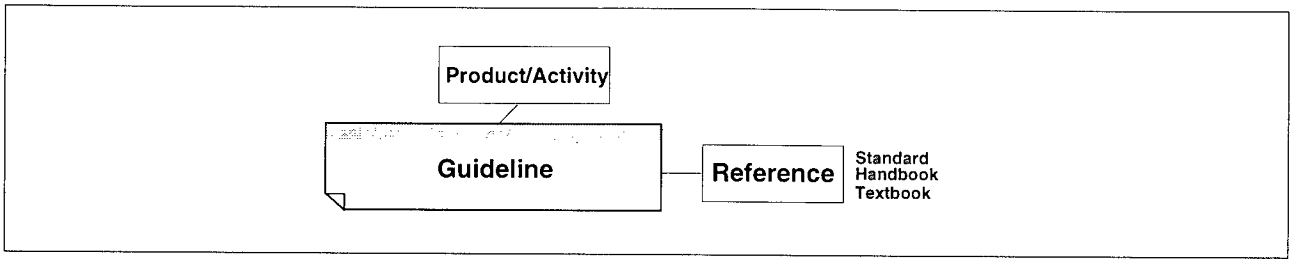


Figure 2-7. Linked to Product/Activity and Reference

Figure 2-8 illustrates an example of guidelines derived from questions for the Accuracy criterion. The arrow pointing from the reference block of the lower guideline to the upper guideline indicates linkage resulting from the flow down from one guideline to another in later phases of the development life cycle. This linkage is also useful for tailoring. If for example, the higher-level guidelines are tailored out, the lower-level, or derived, guidelines can be easily identified for tailoring by the linkage relationship.

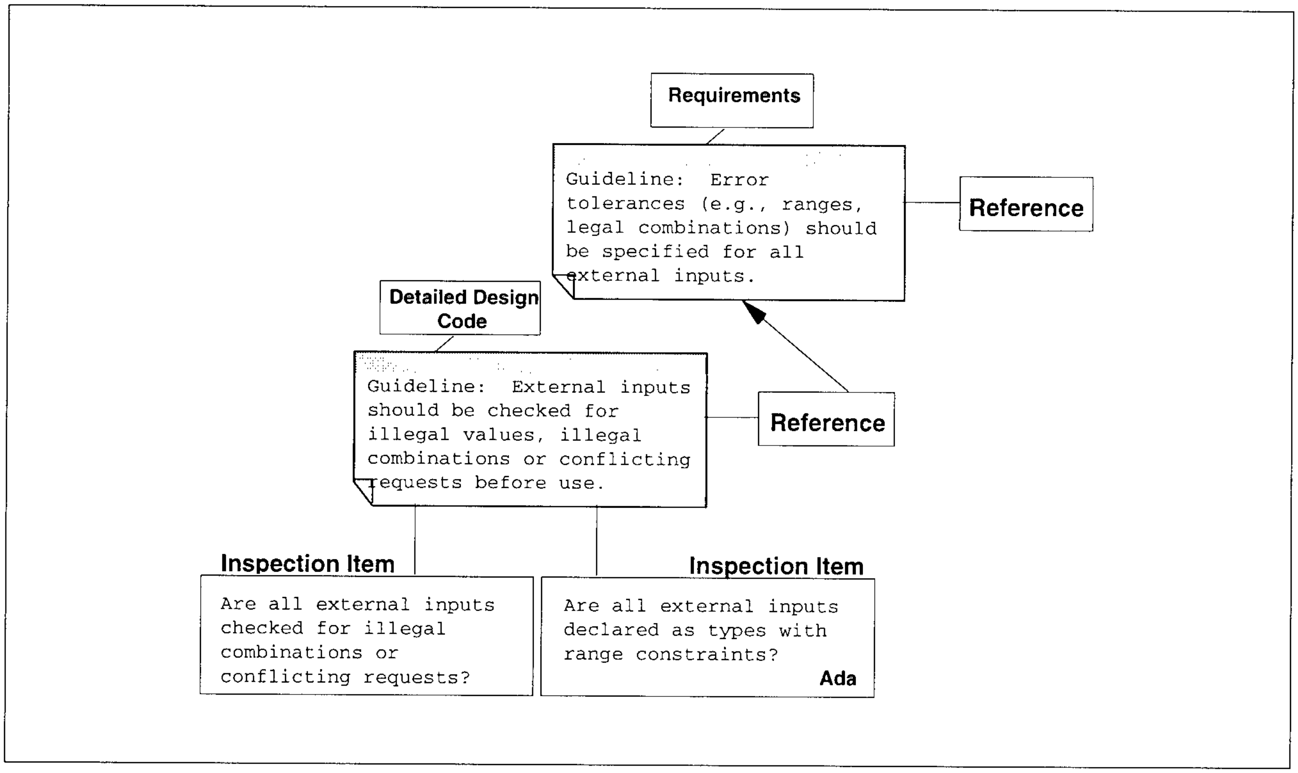

Requirements
Guideline: Error tolerances (e.g., ranges, legal combinations) should be specified for all external inputs.
Reference
Detailed Design
Code
Guideline: External inputs should be checked for illegal values, illegal combinations or conflicting requests before use.
Reference
Inspection Item
Are all external inputs checked for illegal combinations or conflicting requests?
Inspection Item
Are all external inputs declared as types with range constraints?
Ada

Figure 2-8. Example Guidelines for the Accuracy Criterion.

2.4.2 Guidelines Examples

Examples of guidelines extracted from the SQF are provided for three criteria, Accuracy, Simplicity, and Generality. This reverse-engineering approach could be used to turn the SQF into a handbook.

Accuracy deals with those characteristics of software which provide the required precision in calculations and output. There are 25 questions for the criterion Accuracy in the 1991 version of the SQF, and they are distributed over all 9 phases. They can be summarized by the following guidelines.

Accuracy Guidelines

An error analysis should be performed prior to allocating requirements to CSCIs. Accuracy requirements should be budgeted to individual capabilities. There should be quantitative accuracy requirements for all inputs, outputs, and constants associated with mission critical capabilities. Accuracy requirements should be checked for traceability down to the CSU level. At all levels of testing (unit, CSC, CSCI, and system) outputs should be checked to ensure that they meet the accuracy requirements. Reused code such as math libraries should be evaluated to ensure that it provides adequate precision.

Simplicity embodies those characteristics of software which provide for definition and implementation of functions in the most uncomplicated and understandable manner. Simplicity guidelines are more detailed, and are less succinctly summarized than Accuracy guidelines, especially at the detailed design and coding stages. There are 85 questions in all for the criterion Simplicity.

Simplicity Guidelines

Capabilities should be identified in a structured fashion using diagrams. There should be a requirement for a programming standard, and a programming standard should be established as early as the system requirements analysis phase. There should be a requirement to use a structured language (such as Ada) or a preprocessor. Code should be checked to see that it follows the programming standard.

The design should reflect a structured approach. Descriptions should identify all interfacing units and hardware. The use of common blocks should be minimized. Repeated or redundant code should be avoided by using language features such as procedures, etc.

Each unit should perform a single, non-divisible capability, and be self-contained (i.e., independent of the source of input and destination of output, and of knowledge of prior processing). Its description should include input, output, processing, and limitations. The flow should be simple from top to bottom, with one entrance and one exit, minimal branching, no nesting of levels or unnatural loop exits. Negative and compound Boolean expressions should be avoided. The unit should be free of self-modification of code.

Each unit should reference a minimal amount of data items and each data item should have a single use. Use of global data items should be avoided. The number of input data items should be minimized and output should be done via parameters. The number of unique operands should be minimized.

Generality deals with those characteristics of software which provide breadth to the functions performed with respect to the application. Generality guidelines can be summarized from the 32 questions found throughout the framework. Note that the guideline not to range-check inputs is in direct opposition to the Accuracy guidelines.

Generality Guidelines

The system should be free from machine-dependent operations. Components (e.g. CSUs) should be designed to perform single processing capabilities. There should be no strict limits on the volume of data handled by the system. The range of input data values should be flexible (i.e., there should not be hard-coded error tolerances or range tests.)

CSUs should be designed to be called by more than one other CSU, and should contain a mixture of external input, external output, and algorithmic processing.

2.4.3 Guidelines and Configuration Management

Two of the challenges of management of the framework are to determine what is there, and to evaluate the desirability of proposed changes. Evaluators may find it easier to understand and interpret the framework in the form of statements, rather than as a collection of questions.

When examining the guidelines, we often find that we do not necessarily agree with all of them. For example, the Anomaly Management question,

```
AM.1.5.e     When an error condition is detected, is its
resolution determined by the calling CSU?
```

when turned into a statement leads to the following guideline:

Guideline: All error conditions should be resolved by the calling CSU.

This approach may not be desirable for all designs. For example, if a well-defined part of the architecture is responsible for handling user error messages so that all messages appear the same to the user and are consistently handled, then this would violate the guideline. In Ada, for example, units might handle different types of errors in different ways. A unit might immediately resolve a low-level predefined exception such as "Constraint_Error" and/or convert it into a meaningful application-defined exception and propagate it to the calling body. It might detect an incorrect request, or incompatible inputs, by raising its own exception. A data entry unit might deal with an incorrect keystroke by simply BEEP!ing and ignoring it.

If we examine the Phase C version of this question, we are inclined to derive a slightly less stringent guideline.

```
AM.1.5.c     Is there a standard for handling recognized
errors such that all error conditions are reported (via
raising, propagating exceptions, or passing a value) to the
calling body (e.g., subprogram, task or package)?
```

Guideline: There should be a standard for handling recognized errors such that all error conditions are resolved immediately or are reported to the calling unit for resolution.

Recording references and dependencies for each guideline will also be helpful for framework management. When a standard changes or is made obsolete, the related guidelines will be easily isolated for updating. The fact that a reference exists at all is an indicator of the acceptability of the guideline. Dependency information may be used to derive different versions of the framework, such as an Ada version, or an object-oriented version.

2.4.4 Analysis of Software Development Life Cycles

In order to develop an approach for redesigning the SQF to improve its tailorability to non-Waterfall development life cycle processes, we first analyzed common development life cycle models in use today. This section begins with an overview of the common models and concludes with the results of our comparative analysis.

2.4.4.1 The Waterfall Model

The Waterfall model officially appeared around 1970 [ROY70], but earlier versions were described as early as the late fifties. This model has direct ties to computer hardware engineering. The Waterfall model depicts the software development process as a series of ordered, sequential "phases" as shown in Figure 2-9. The input for each phase is a documented specification from the previous phase. The focus of each phase is to *translate* these input specifications into a more concrete, less abstract specification of the system. Early phases deal with the problem space and later phases with the solution space.

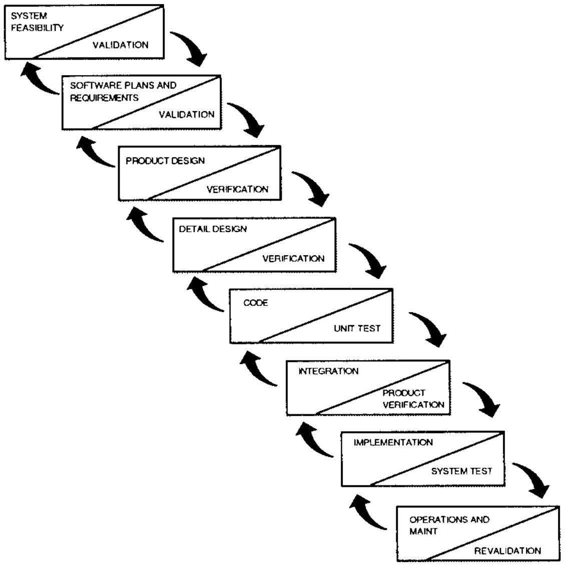


Figure 2-9. The Waterfall Model

The Waterfall model has its advantages. Software personnel can *specialize* in a certain phase of development—for instance, an analyst can focus on defining requirements, a designer on software design, a programmer on coding, and so on. Also, the process is simple, because of the sequential nature of the work, and the interfaces between phases and people are straightforward -- the producer of the completed phase hands off the project and a complete specification to the producer of the next phase. This model has a built-in requirement for documentation at each level which is beneficial because, theoretically, all decisions are documented and traceable.

Experience, however, has uncovered a number of problems with the Waterfall model. One problem is that customers are typically only involved at the front-end (requirements analysis) and back-end (testing) of the software development process. This lack of involvement often results in a system implemented with wrong, missing, and extra/unnecessary requirements. (Imagine the Waterfall as an implementation of the game "telephone" where developers have little opportunity to get clarifications on unclear messages, resulting in extremely garbled messages coming out the "back end" of the pipeline.) This lack of involvement also results in lack of ownership and acceptance of the final system on the part of the customer community.

While later Waterfall variations attempt to deal with flawed specifications by building in feedback loops to earlier phases, most allow the loop to span only one phase, leaving no mechanism for dealing with new requirements uncovered downstream. Experience has shown that it is often impossible to get complete requirements up front. Therefore, when new requirements surface very late in the life cycle (i.e., testing or maintenance), they are hard to deal with and much more costly to fix [BOE81]. Finally, one of the most glaring problems is that the model's linear nature results in a process that takes too long to complete; by the time the system is developed, requirements have changed and the system is often inadequate, or worse, irrelevant.

2.4.4.2 The Incremental Development Model

The Incremental Development model, shown in Figure 2-10, is a slight variation of the Waterfall. After the system-level specification phases have been completed, a series of mini-projects are started. While each mini-project's process remains serial, this strategy allows the mini-projects to proceed in parallel. This can help reduce the lengthy cycle time problem associated with the Waterfall, but can also create *more* product problems due to the large number of interfaces that now must be coordinated and controlled between mini-projects.

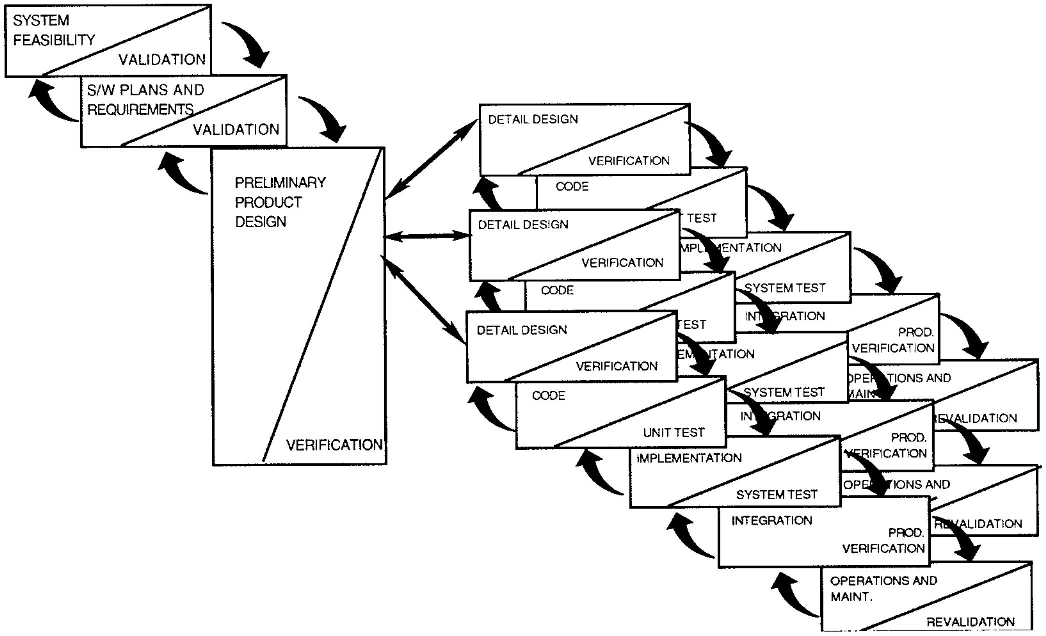


Figure 2-10. The Incremental Development Model

Another variation of the Incremental Development Model is to re-integrate the mini-projects into a single project for the later phases involving testing, implementation, and operations/maintenance.

2.4.4.3 The Spiral Model and its Variations

The Spiral model is probably the second-most-familiar model next to the Waterfall. This model was initially proposed by Boehm in 1975, with various refinements made in 1976, 1986, and 1988 [BOE88]. The characteristic shape is shown in Figure 2-11. The concept of the spiral model is that you start from the middle of the spiral and work your way out. Each cycle or rotation, takes the developer through the same four steps (i.e., shown as the four quadrants in the picture):

1) determining objectives,
2) evaluating risks/alternatives,
3) developing/verifying, and
4) planning the next cycle.

Each cycle addresses a further level of decomposition or "phase", which can be seen by looking at the activities shown as spirals in the lower right quadrant. The process is risk-driven, in that the details of what is actually developed or planned in the subsequent steps is based on the risks identified during that cycle.

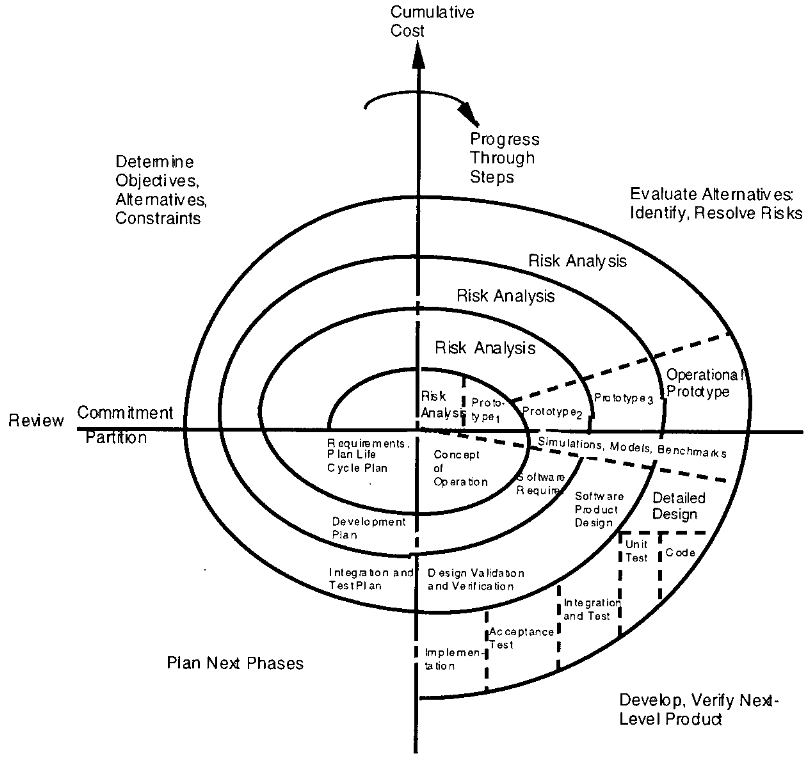


Figure 2-11. The Spiral Model

The risk-driven nature of the model adds flexibility and helps support whatever approach (e.g., evolutionary, specification-driven, prototyping, etc.) is best suited to the problem at hand. For instance, prototyping is an option in each cycle, but is not required. (Prototypes may be written to verify the user interface, performance, or to evaluate design alternatives). Written specifications are also optional; if an evolutionary prototyping approach is used, the verified prototype and a written plan may serve as the "specification" for the next cycle.

The spiral model has encountered some criticism. One complaint by users is that the model is complex and requires more experienced staff to successfully implement. Also, because the project plans are dynamic and many commitments are deferred until later in the project, it is difficult to use in contract situations where the acquisition process demands plans, costs, and a complete schedule up front.

Various approaches similar to the spiral model have been put forth. Evolutionary Development focuses on the fact that, for many types of systems, the requirements simply cannot all be determined up front, and the execution of the software development process actually uncovers unknown requirements that may not be discovered otherwise [MCC82]. With this approach, a small piece of operational software (sometimes called an operational or evolutionary prototype) is developed, reviewed, refined, then "added onto" in the next iteration. The Iterative Development model [KAN95], illustrated in Figure 2-12, is another example.

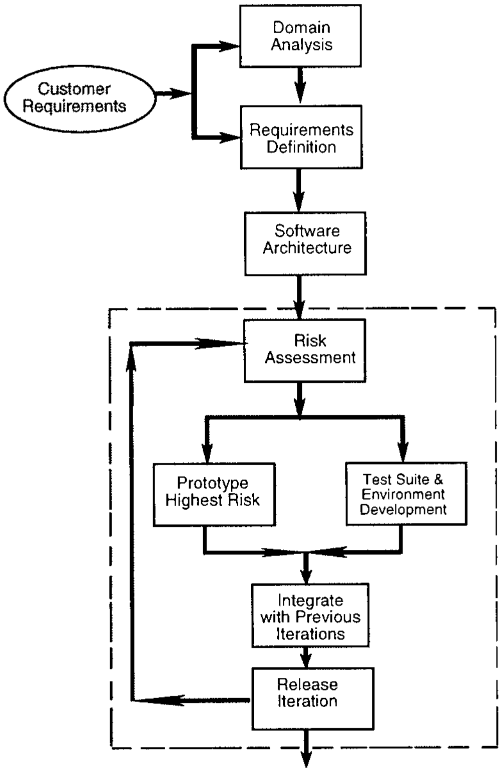


Figure 2-12. The Iterative Development Model

All approaches which involve iterations and prototypes require continuous customer involvement to review the operational prototypes and refine requirements. A noted problem with approaches involving evolutionary prototypes is that the code-and-fix cycles can result in poor structural quality. A large number of fixes, over time, can weaken design at the system and code/unit level. Rapid code-and-fix cycles can also cause developers to loose sight of the big picture, causing unnecessary requirements and design changes to insidiously creep into the system.

2.4.4.4 The RAD Approach

Rapid Application Development (RAD) appeared on the scene in the mid 80's, in response to the need to reduce software development cycle time. While no agreed-upon RAD model has ever been adopted in the industry (each vendor has their own), most RAD approaches share a set of common characteristics. They emphasize timeboxing of activities (shown in Figure 2-13), facilitated requirements workshops, the use of prototyping and CASE tools, and strict scope control.

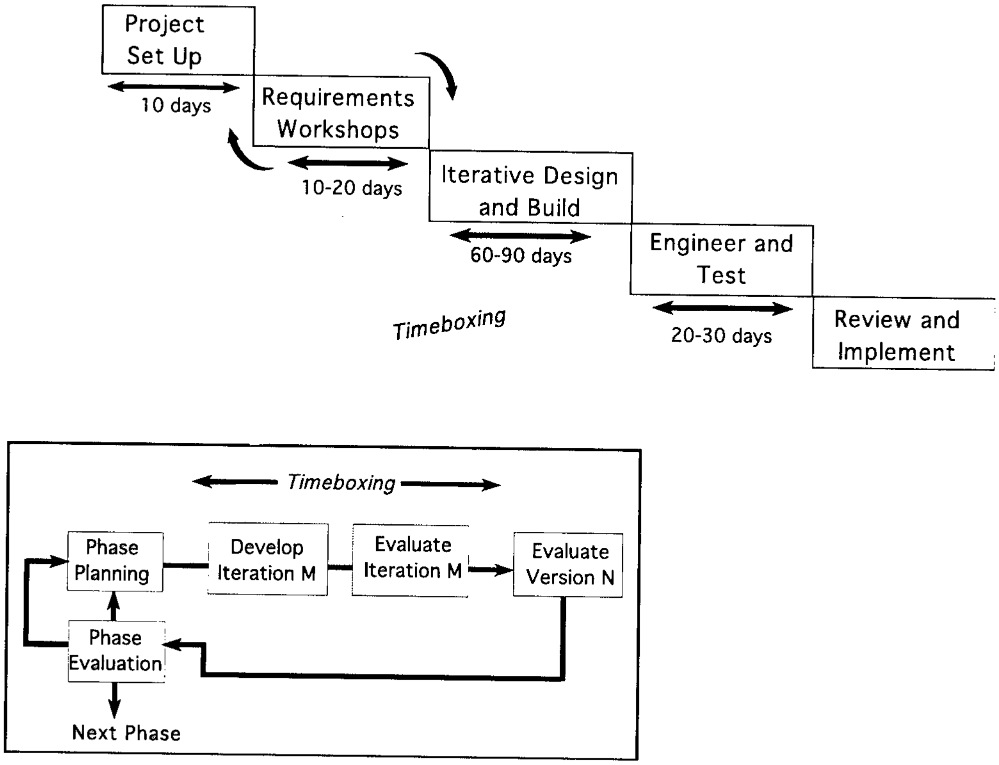


Figure 2-13. Timeboxing in the RAD Approach

Proponents of this approach point to the ability to deliver systems typically in 6-9 months versus 2-3 years, and claim that these systems better meet business needs because of the close customer involvement that RAD fosters. Critics claim that many systems are unsuitable for RAD and that the approach actually takes much longer than proponents claim, because scope creep is extremely difficult to control. Another criticism is that design is often skipped or de-emphasized, leading to "brittle and unmaintainable" systems [REI95]. While RAD may never become a recognized life cycle model, it can be said that most of the techniques which help characterize this approach are considered commonplace in many IT organizations.

2.4.4.5 OO Life Cycle Model

While Object-Oriented (OO) technology is not new (it's roots actually go back into the 70's), the notion of a new paradigm for software development based on a body of OO methods, processes, and tools emerged in the 90's. Like RAD, no agreed-upon, standardized depiction of an OO life cycle model exists. However, most OO experts would agree that the OO life cycle is similar to traditional life cycle models in that the same activities are still performed (analysis, design, coding, testing, etc.), yet is different from traditional life cycles in a few major ways.

First, the nature of OO development lends itself to more iteration and overlap than other models. In part, this is because while traditional life cycles start with and emphasize functional decomposition and are basically procedure-driven, the OO approach stresses instead the encapsulation of data and procedural features together (objects). In this way, both high-level analysis and design are accomplished in terms of these objects and the services they provide to the user. With the OO model, the transition from requirements to design is not a transition from a functional view of the problem space to a structural view of the solution space, but rather, a transition from objects in the problem space to objects in the solution space. Also, because the reuse of objects is a key objective, the paradigm shifts from top-down to bottom-up for the detailed design and implementation stages of the OO life cycle. This concept is illustrated in the "fountain" model of Figure 2-14.

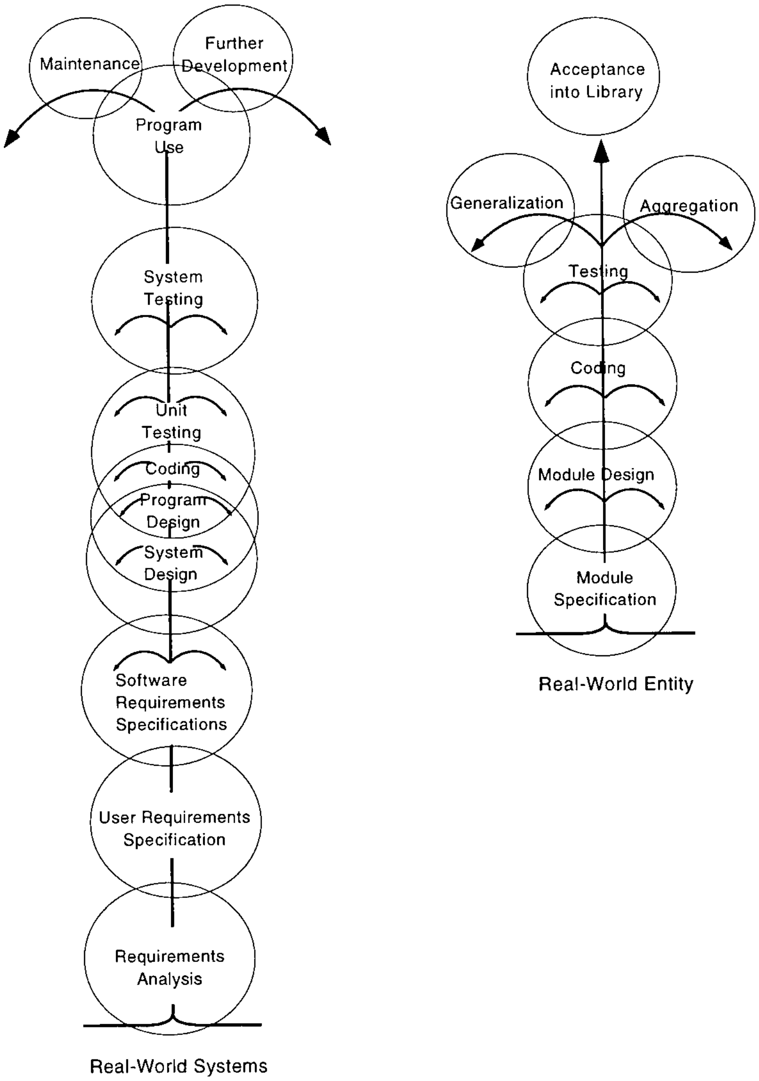


Figure 2-14. OO Fountain Model

Placed in the timeline context, the OO Life Cycle looks something like Figure 2-15. Requirements and Design grow over time as decentralized clusters of objects are implemented and refined [HEN90].

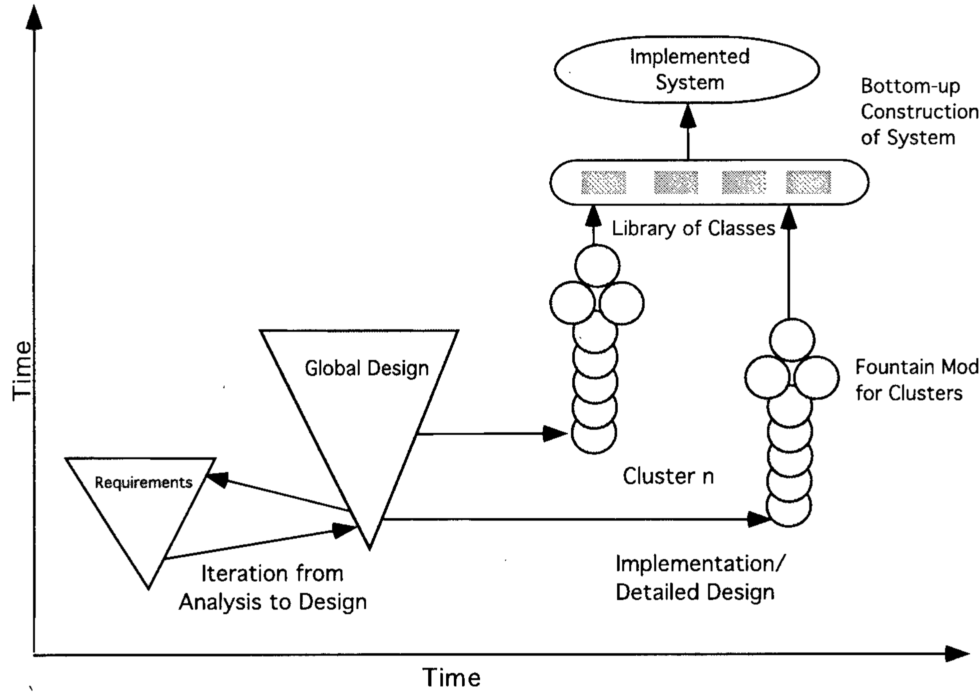
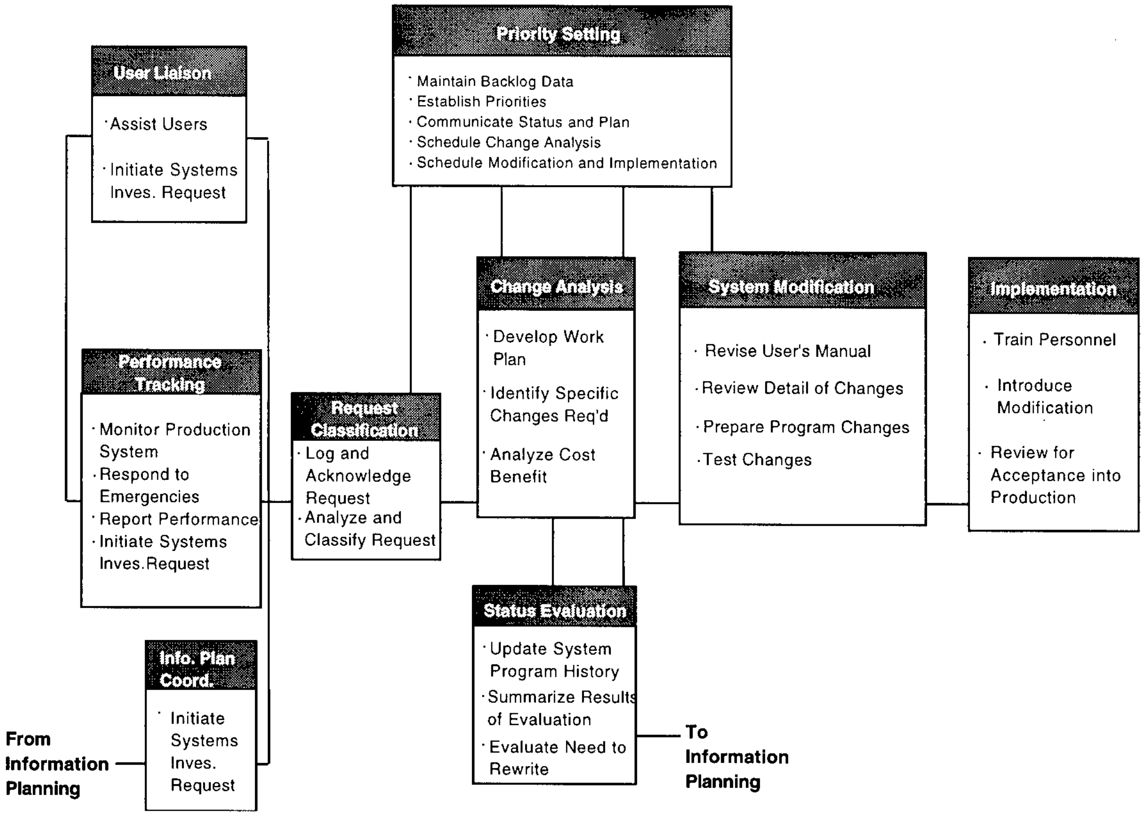

Implemented System
Bottom-up Construction of System
Library of Classes
Fountain Model for Clusters
Global Design
Requirements
Cluster n
Iteration from Analysis to Design
Implementation/ Detailed Design
Time
Time

Figure 2-15. OO Life Cycle

2.4.4.6 Maintenance Life Cycle

We use the term maintenance to refer to changes that have to be made to software after it has been delivered to the customer [MAR83]. Many of the life cycle models reviewed here include a phase labeled "maintenance", however, the maintenance phase of the life cycle itself can also be considered a type of software process model. While it is often looked at as a miniature version of the software development process, in reality, there are differences. The maintenance process needs a quick, yet controlled approach for responding to emergencies which affect business operations; there is also the problem of juggling and prioritizing diverse, unrelated requests from the user community; the process must also deal with the need to continuously assess the impact of changes over time on the operational system's performance, flexibility, and maintainability. Figure 2-16 provides one example of a life cycle model specifically geared to maintenance.



User Liaison
· Assist Users
· Initiate Systems Inves. Request
Priority Setting
· Maintain Backlog Data
· Establish Priorities
· Communicate Status and Plan
· Schedule Change Analysis
· Schedule Modification and Implementation
Performance Tracking
· Monitor Production System
· Respond to Emergencies
· Report Performance
· Initiate Systems Inves.Request
Request Classification
· Log and Acknowledge Request
· Analyze and Classify Request
Change Analysis
· Develop Work Plan
· Identify Specific Changes Req'd
· Analyze Cost Benefit
System Modification
· Revise User's Manual
· Review Detail of Changes
· Prepare Program Changes
· Test Changes
Implementation
· Train Personnel
· Introduce Modification
· Review for Acceptance into Production
Status Evaluation
· Update System Program History
· Summarize Results of Evaluation
· Evaluate Need to Rewrite
Info. Plan Coord.
· Initiate Systems Inves. Request
From Information Planning
To Information Planning

Figure 2-16. An Example of a Maintenance Life Cycle

2.4.4.7 Comparative Analysis

Some things become apparent when these various life cycle models are reviewed together in this way. The conclusions of our comparative analysis are as follows:

- **The set of activities needed to produce any software product and manage the corresponding software project are similar.** There is a common set of activities associated with software development and maintenance. While each life cycle may use a different name and description of the activity, they are still more similar than different. For instance, every life cycle uses the notion of requirements analysis, design, implementation, and verification.

- **The order, duration, concurrency, and iterative nature of activities is different depending on the objectives of each model.** In other words, the ways the activities are fitted together are different. For instance, while requirements analysis is a task with a fixed order and duration in the Waterfall model, it is iterative in the evolutionary development model.

- **For each common activity, there are countless way of accomplishing the activity**. Methods, tools and techniques constrain and determine the way work is accomplished. For example, requirements can be gathered using either a facilitated meeting, focus groups, or structured interviews. Code can be "implemented" by selecting code from a reuse library, generating it from a CASE tool, or by writing a series of instructions from scratch.

- **For each common activity, the outputs (or products) generated share a set of required attributes and are used similarly as inputs to other activities.** While the result or output of a common activity may take a different form depending on the methods, techniques, tools used, a common set of desired attributes can still be verified.

As a result, we developed a "building blocks" approach to tailoring the SQF to various life cycle models. This approach is discussed in the next section.

2.4.4.8 Building Blocks Approach

Our re-engineering approach to tailoring the SQF application to various life cycle processes is based on the following principles:

- Adopt standard terminology in the framework for software products and activities (the "building blocks")
- Link framework guidelines and indicators to these standard products and/or activities
- Tailor by selecting guidelines and measures relevant to products and activities applicable to the life cycle process
- Merge the resulting SQF inspections and measurements into the life cycle process.

Standard terminology was adopted from the two publications listed below. The two efforts which resulted in these publications acknowledged the similarities among the different life cycle models and attempted to provide guidance which encompasses all of the various life cycle models. The purpose of both publications is to define a set of mandatory activities for the development and maintenance of software, regardless of the particular life cycle used. They are:

- IEEE Standard for Developing Software Life Cycle Processes (Std 1074-1991)
- U.S. MIL-STD 498 (Software Development and Documentation)

MIL-STD 498 defines 18 common activities and uses Data Item Descriptions (DIDs) to specify a set of associated output products or deliverables that should be produced as part of any software development effort. The standard also contains some good examples of how the standard could be applied to different development strategies. IEEE 1074 defines 17 processes and 65 activities. While the names of activities and processes in these standards differ, there is much overlap in terms of the process "building blocks" defined. Each activity in 1074 is described by 1) inputs, 2) actions to be performed, and 3) outputs generated. The outputs of an activity are the measurable or inspectable "products". Unlike the DID approach used in MIL-STD 498, the output information contained in this standard does not specify the production of any specific deliverable, but rather focuses on the information that needs to be passed between activities.

Examples of the activities and products are listed in the following table.

Example Activities	Example Products
Implement Problem Reporting Method	Corrections Problem Reported Info.
Concept Exploration	Statement of Need
System Allocation	Recommendations (feasibility studies)
Requirements	System Architecture
Design	Software Interface Requirements
Implementation	Software Requirements
Plan V&V	Software Architecture
Execute the Tests	Software Design Description
	Source Code
	Test Planned Info.
	Test Summary Reported Info.

2.5 Re-engineered SQF

This section describes the results of the SQF Re-engineering task.

We will refer to the last version of the SQF prior to this re-engineering task as Version 1.5. Figure 2-17 illustrates the history of the SQF leading to Version 1.5. It is essentially similar in structure and in method of application to the two major versions that preceded it [MCC77], [BOW85]. Version 1.0 was created as part of the Quality Evaluation System (QUES) development contract which incorporated the DOD-STD-2167A terminology from the STARS Software Evaluation Report DIDs, added Ada-specific examples, and extended the scoring equation methodology for tool automation.

The SQF was placed under configuration management and given version designations as part of SPS' QUES maintenance contracts for Rome Laboratory because the framework was delivered in database form as part of the QUES software deliveries. The version numbers match the QUES software version numbers; changes from Version 1.0 through 1.5 were only minor corrections to question text and to scoring equations.

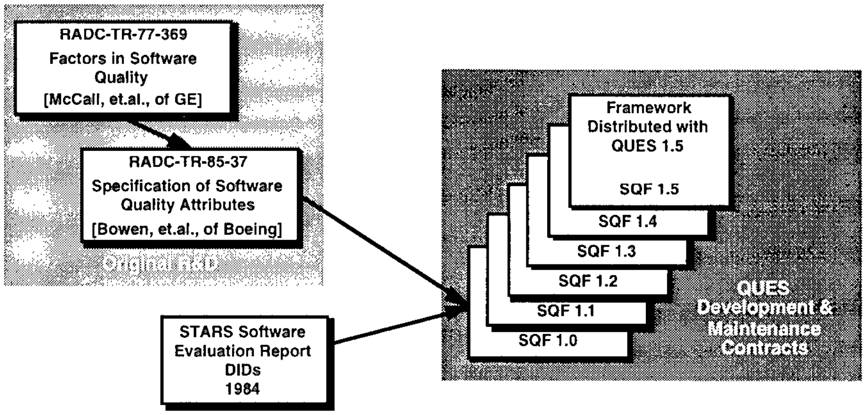

RADC-TR-77-369
Factors in Software Quality
[McCall, et.al., of GE]
RADC-TR-85-37
Specification of Software Quality Attributes
[Bowen, et.al., of Boeing]
Original R&D
STARS Software Evaluation Report DIDs 1984
Framework Distributed with QUES 1.5
SQF 1.5
SQF 1.4
SQF 1.3
SQF 1.2
SQF 1.1
SQF 1.0
QUES Development & Maintenance Contracts

Figure 2-17. Origin of SQF Version 1.5.

A new version constructed from this re-engineered design would be designated 2.x. We have not constructed a complete version 2.0 in this effort, but have prototyped a slice of the framework for the factors of Portability (see section 2.5.5) and Correctness (see section 3.4) to illustrate the concepts.

2.5.1 Re-engineering to Improve Adaptability

Our objective in re-engineering the SQF is to address some of the deficiencies of the previous design in terms of usability, adaptability, understandability, and completeness. An important characteristic of any framework is its ability to adapt to a variety of real-world software development situations.

There are three main adaptability concerns that comprise different points of view into, or dimensions of, the framework's structure:

- quality factors
- life cycle processes
- software products

Quality Factors Dimension

The SQF quality factors represent different, altł.ough not orthogonal (i.e., independent), desirable characteristics of a software product. They can be thought of as various aspects or facets of quality, or as a partitioning of quality-related requirements. The current set of 13 factors is meant to be applicable to a broad range of software-intensive systems in many domains.

Recognizing that not all systems have the same quality requirements to the same degree, factors are selected and ranked in order of priority. This allows developers and customers to discuss the relative importance among the requirements as a prelude to making trade-offs. Some factors, for example, are known to be opposing in that optimizing one will have a negative impact on the other. Trade-offs among other factors, while not opposing, simply reflect the need to prioritize the application of limited resources.

Measurable attributes of the software products are linked to factors via the factor-criterion-metric hierarchy of the framework. This permits tailoring of quality evaluation to the selected factors of interest.

Life Cycle Processes Dimension

The software development life cycle is a temporal view of the steps of a development process. The SQF was originally based on the waterfall life cycle model, and Version 1.5 uses the DOD-STD-2167A terminology. There is one Version 1.5 data collection form for each life cycle phase, and the architecture level at which the data is collected varies as shown in Figure 2-18. The simplicity of the waterfall model makes it easy to understand. Using the waterfall model as a basis for the SQF structure, however, makes it difficult to adapt the framework for use in non-waterfall environments.

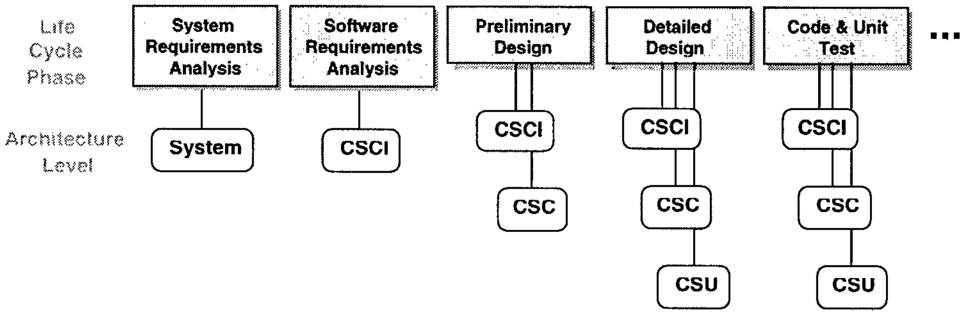


Figure 2-18. SQF 1.5 Data Collection Designed for Waterfall Process.

The purpose of the life cycle aspect of the framework structure is to convey *when* quality evaluations are to be performed, and to show how the nature of the evaluation changes over the course of the life cycle. The main reason that the nature of the evaluation changes is that the software products vary as discussed in the following section.

Software Products

As the software development life cycle progresses, products are created that convert abstract concepts into more concrete representations. As these representations approach the software system in its final form, our ability to evaluate the quality of the software improves. Early in the life cycle, quality evaluations are predictive, in that they can only indicate the quality of the end product by assessing the quality of intermediate products. It may not be until the system is fully operational in its target environment that its quality can be truly assessed.

Early life cycle evaluations are also mainly heuristic or theoretical, because they are based on assumptions and guidelines that may not have a proven link to the quality of the end product. Having a complete, robust set of requirement specifications, for example, is a good start, but is no guarantee that the product will actually fulfill those requirements. Despite these limitations, the SQF emphasizes early life cycle evaluations because the earlier a quality problem is found, the lower the cost to correct it.

Software products, such as requirements specifications, architecture, design, code, test plans, and documentation, are outputs of activities performed during the life cycle, and their size is an indication of the amount of work performed during that activity.

2.5.2 Re-engineering the Organizational Structure

The SQF's organizational structure is the skeleton of the framework. Like any framework, it provides an organized way of thinking about the problem—in this case, evaluation of software quality. As discussed in section 2.5.1, the structure has three basic dimensions:

- quality factors
- life cycle processes
- software products

The quality factor dimension addresses *what* to evaluate, the life cycle dimension addresses *when* and *how often*, and the product dimension addresses *how much*, or what portions, of the system to evaluate.

The life cycle processes and products dimensions are linked by the association of a product with one or more activities. If a product is either created by or modified by an activity, then the product's measurement is associated with that activity. Measurement should be done *each time* that activity is performed, and should be *as of* the completion of that activity. Thus the activity provides a "time stamp" for the measurement.

Adaptation of Life Cycle Models

By viewing software life cycle processes as comprised of the basic building blocks of activities and products, it is possible to construct different life cycle models by combining these building blocks as discussed in section 2.4.4. Life cycle models (such as waterfall, modified waterfall, incremental, and evolutionary) are differentiated by which activities are included, how the activities are ordered, and what portion of the system is addressed at a time. By tying quality guidance, inspections, and measurements to these basic building blocks, the quality framework is then applicable to a wide range of life cycle models.

Instrumentation for Quality Measurement

Evaluation of product quality requires up-front planning in order to have the mechanisms in place to measure quality throughout the development life cycle. The SQF should assist the planner by helping to identify what measurements will be needed, and determine how to instrument the development process. Proper instrumentation ensures that these measurements will be performed during the right activities, at the right time, and on the right products.

In general, the more closely integrated the measurement is with the development process, the more valuable the results of that measurement are likely to be. The value of a measure is in its link to decision making and corrective action; if it is taken too late or reported too infrequently to have any impact on the project, it is of no value other than historical interest.

It is not enough to specify a measurement on a product without discussing issues such as what constitutes a "unit" of observation (e.g., module, class, package, subroutine), when a unit is considered complete enough to be measured, and how the measurement process is affected by the rework of a unit.

Activity versus Product Measures

Activity-based measures are usually associated with project management and process improvement viewpoints. Typical activity measurements are duration (start and end dates), effort expended, and some measure of the amount of work performed. The amount of work performed is used for assessment of both progress toward completion and productivity. Estimates and actual measures are compared to assess the accuracy of estimates. From the activity view, defects are tracked by the activity where inserted and where detected, for causal analysis and detection efficiency analysis.

Product measures are usually associated with quality concerns. These typically fall into the categories of structural size & complexity, defects, growth & stability, conformance to standards, and performance. Size & complexity measures serve several purposes, such as inputs to effort estimation, comparison of design alternatives, and isolation of problem areas. Defect counts are typically normalized by computing density to allow comparison with historical norms, and to look at trends over time.

Product growth & stability measures look at the change in size over time, also comparing estimates to actuals, to identify uncontrolled growth in scope or rework risk. Products are inspected with respect to standards, and violations are treated as a type of defect (with the main variation being the severity associated with that type of defect). Performance measures are specific to performance requirements, and are associated with simulations, prototypes, or tests.

2.5.3 Re-engineered Quality Factors Hierarchy

In the SQF Version 1.5 design, the factor-criterion-metric hierarchy is used to translate abstract user-oriented goals into concrete, measurable attributes of the software. In our re-engineered design, the hierarchy still serves the same purpose, but has been modified as shown in Figure 2-19. We have retained the top two levels of the hierarchy, the factors and criteria. At the third level, we have subdivided the framework into two types of information: guidelines and indicators. Rather than adding new information to the old framework, instead, this simply reorganizes what was already there into a more useful form.

This new split in the hierarchy makes the "measurement framework" part of the SQF stand out from the part which is essentially geared to guidance and product inspections. The whole SQF can be thought of as a "quality framework" for software because it contains prescriptive guidance about how to build quality in, coupled with verification techniques (i.e., product inspections) to assess the achievement of quality goals.

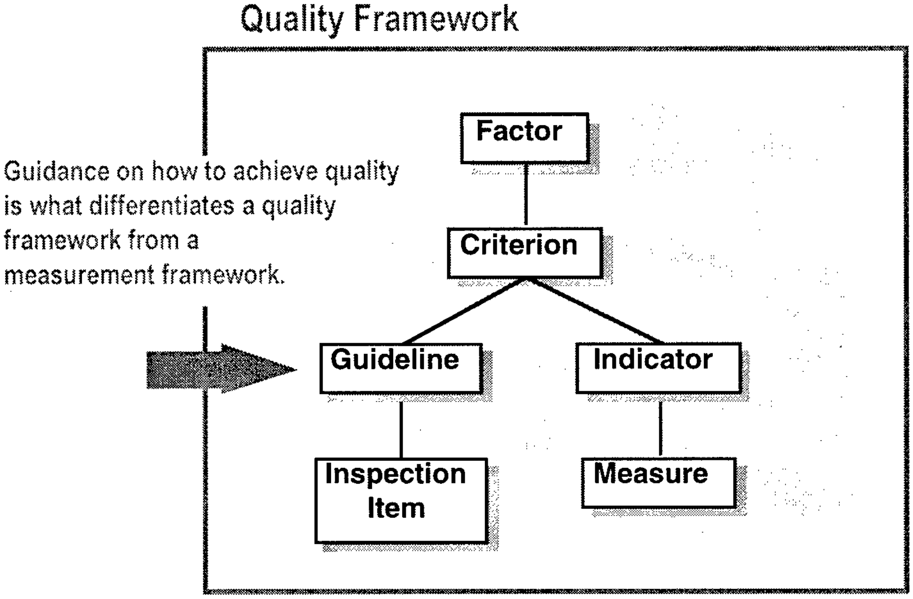

Quality Framework
Factor
Criterion
Guideline
Indicator
Inspection Item
Measure
Guidance on how to achieve quality is what differentiates a quality framework from a measurement framework.

Figure 2-19. New Quality Factor Hierarchy.

Guidelines

The guidelines that exist in SQF 1.5 are implicit in the questions found on the data collection forms. These implicit guidelines vary in their degree of validation, from proven software engineering best practices to essentially theoretical ideas. Some assume particular design paradigms such as structured analysis/structured design.

We believe that it is better to present the guidelines as straightforward statements, to provide references that establish the degree of validation, and to explicitly identify assumptions.

Some guidelines are conceptually applicable throughout the life cycle, but their manifestation changes to reflect the nature of the activities being performed. The basic pattern is outlined in the following table.

Life Cycle Phase						
Concept Exploration	Requirements Analysis	Design	Implementation	Testing	Deployment	Maintenance
Evaluate the need for X	Specify a requirement for X or incorporate X into plan	Incorporate X into the design	Implement X	Test for X	Evaluate/ measure X	Same as previous for new development activities

The earlier SQF approach focuses on converting a quality concern into testable requirements. From there it can be treated as any other requirement, using the requirements tracing technique to verify that the requirement is manifested in the implementation and confirmed through testing. This approach assumes that the developer already has an adequate process in place to achieve his requirements.

Our new SQF redesign incorporates this earlier requirements-driven approach and supplements it with guidance and indicators.

For example, a guideline pertaining to the desirability of modular design would be to specify modularity in a requirement, plan, or development standard. During design, the guideline would describe characteristics of a modular design and recommend evaluating modularity in design inspections and reviews. After detailed design and implementation, the modularity achieved could be directly assessed.

New Measurement Approach

The Version 1.5 approach to measurement was to compute a value for each quality factor based on complicated (and unvalidated) scoring equations containing a mixture of inspection results and product measures. Our redesigned approach simplifies measurement by eliminating the Version 1.5 scoring equations and not "measuring" inspections. Instead, one or more indicators are used which are based on product measures and are evaluated individually rather than as part of one massive equation.

Indicators are direct or indirect measures of properties that are demonstrably related to a factor, criterion, or guideline. Even though Figure 2-19 shows the Indicator level of the reengineered hierarchy to be linked to the Criterion level, indicators can be designed for any level as shown in Figure 2-20. Our term *indicator* is analogous to the common usage of the term "metric". Examples of indicators are requirements traceability, design complexity, defect density trends, and work unit progress.

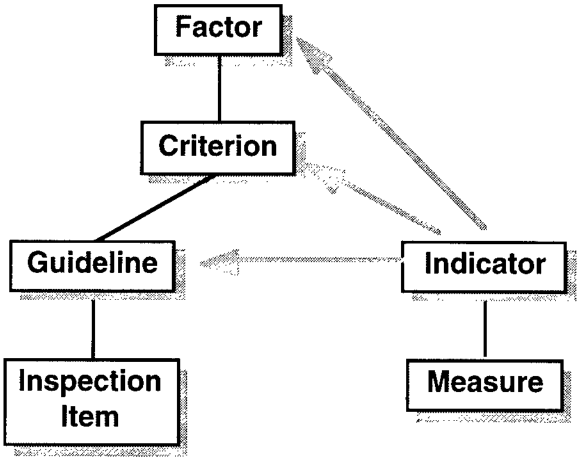


Figure 2-20. Indicators Can Be Designed for All Hierarchy Levels

2.5.4 Re-engineered Tailoring Approach

Most project-level quality efforts today focus on verifying and validating *functional* quality and controlling those few other attributes of software quality that can be easily assessed or measured (e.g., complexity). Unfortunately, these attributes are not always the attributes that are critical to mission success. Projects need guidance uncovering a system's *true* quality requirements. They need a well-defined software quality vocabulary, a way to specify quality requirements that are meaningful to the customer, and a way to translate those requirements into design and implementation specifications. Additionally, projects need a way to then proactively build these quality requirements into the software, rather than relying on testing late in the life cycle to identify quality problems.

Projects also need a way to assess how much quality they need and how much time and effort is needed to deliver that "amount of quality". Just like functional requirements, quality requirements have a cost associated with them, and tradeoffs must often be made in order to meet time and cost constraints. Many factors can influence the cost of building in quality requirements, including:

- the level of quality required (*How* efficient? *How* precise? *How* reliable?)
- the relationships between quality requirements (Can a system be both highly efficient and highly maintainable?)
- where in the life cycle quality requirements can be assessed
- whether automation is available to help assure quality

Having answers to these questions allows the project to make informed tradeoffs to better optimize project resources while meeting project goals.

We have developed a flexible front-end "tailoring process" which uses the re-engineered SQF as its basis. The output from the tailoring process is an evaluation strategy which can be implemented on a project. This tailoring process simplifies the specification guidance already published [BOW85] and provides examples of how the process could be used by various types of software projects.

General Approach: The Tailoring Process

This section describes a four step tailoring process which can be used by project and software support teams to identify a software system's true quality needs and develop a strategy for assuring software quality 1) during the development or enhancement of the software system and/or 2) throughout the system's life cycle. The process described uses the SQF's software quality model as its basis.

During the tailoring process, illustrated in Figure 2-21, software quality requirements for software-intensive systems are uncovered, specified, and used to develop a customized evaluation plan for a development or maintenance effort. The process includes guidance to help uncover and specify quality requirements, techniques for making tradeoffs among factors and criteria, techniques for relating quality levels to costs over the course of a development effort, and procedures for producing a customizable evaluation strategy for a project or system.

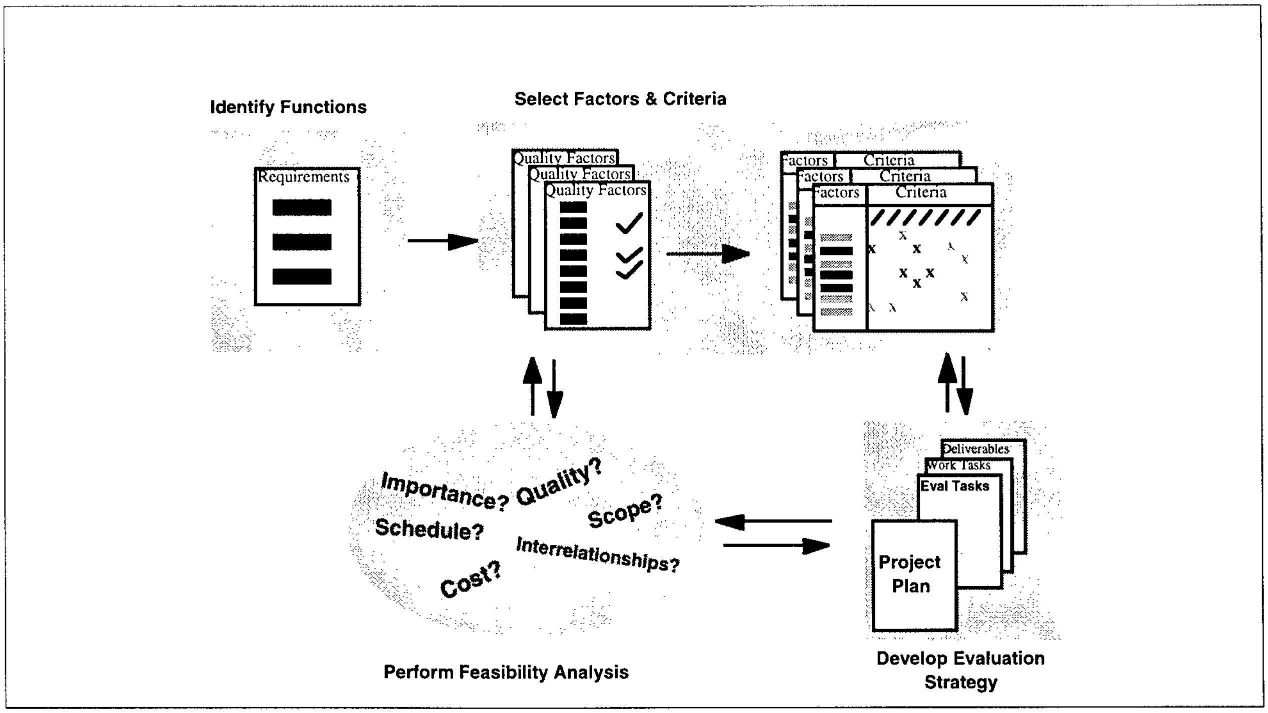


Figure 2-21. SQF Tailoring Process

This tailoring process should be viewed as part of the project planning process. Project plans then drive the development of the product and the evaluation of software quality throughout a system's life cycle. Rather than adding on new process steps, the resulting actions derived from the SQF are merged into the existing software process as shown in Figure 2-22.

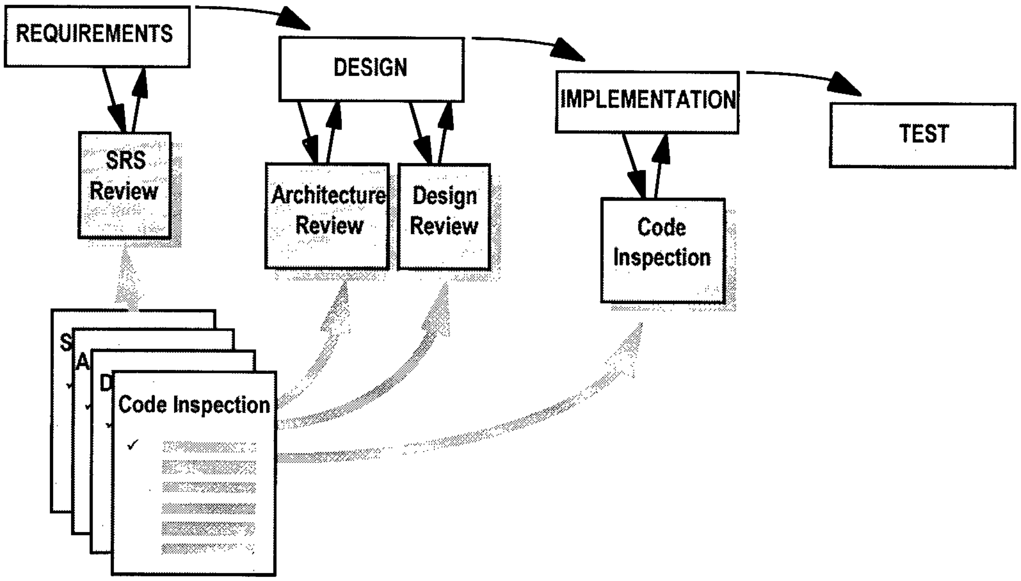


Figure 2-22. Tailoring by Merging SQF Inspections into Existing Process

Step 1 - Identify Functions

Different system functions, and the software supporting those functions, will have different quality requirements. Therefore, the first step in the tailoring process involves partitioning the system into major functional components. Functional components perform a distinct operation such as asset management, communications, order processing, etc. DoD systems are often partitioned into CSCIs representing a system's major functional components. The next step is to identify the functional components that will be supported by software. Each of these functional components will require a tailored evaluation plan.

Next, identify the quality issues or critical success factors for each function. Quality issues are anything that might affect achievement of the function's purpose or mission objectives. Critical success factors are characteristics, within the system developer's scope of control, which must be present if the success of the system is to be assured. For example, critical success factors for the target tracking function of an on-board weapons systems might include the need for a certain degree of precision in locating a target and the ability to provide feedback to the operator quickly.

Critical success factors are often identified during the system and software requirements analysis phases. Risk analysis results are often a source for quality issues.

Each major function component supported by software may have many quality issues or critical success factors.

Step 2 - Select Factors and Criteria

For each major function, map the quality issues and/or critical success factors identified in Step 1, to the SQF's quality factors. Thirteen quality factors have been identified in the SQF's quality model. A single software function may have multiple factors associated with it. Quality factors are user-oriented characteristics that can be attributed to most systems. Some factors are concerned with how well the product performs during operation, others are concerned with how well the software is designed, and still others are concerned with how easily the software can be adapted throughout its life cycle. The words used to describes a function's quality issues or critical success factors may be similar to the quality factors themselves or the words used to describe each factor (see Table 2-3).

Table 2-3. Quality Factor Definitions [SQF95]

Correctness	Deals with the extent to which software design and implementation conforms to specification and standards
Efficiency	Deals with the utilization of a resource
Expandability	Deals with the relative effort in increasing software capabilities or performance
Flexibility	Deals with the ease of effort in changing software to accommodate changes in requirements
Integrity	Deals with software security failures due to unauthorized access
Interoperability	Deals with the relative effort in coupling software of one system to software of one or more other systems
Maintainability	Deals with the ease of effort in locating and fixing software failures
Portability	Deals with the relative effort involved in transporting software to another environment
Reliability	Deals with software failures
Reusability	Deals with the relative effort for converting a portion of software for use in another application
Survivability	Deals with software continuing to perform when a portion of the system has failed
Usability	Deals with the relative effort involved in learning about and using software
Verifiability	Deals with software design characteristics affecting the effort to verify software operation and performance

Knowing how the software will be used, by whom, for how long, and under what conditions is critical to identifying the quality factors that are applicable. Another way to identify applicable quality factors is to consider the environmental characteristics of the software. Table 2-4 identifies some common environmental characteristics and the quality factors most likely to be applicable to functions/systems with those characteristics.

Table 2-4. Examples of Application/Environment Characteristics Related to Quality Factors [BOW85]

Application/Environment Characteristics	Software Quality Factors
Human lives affected	Integrity Reliability Correctness Verifiability Survivability
Long life cycle	Maintainability Expandability
Experimental system or high rate of change	Flexibility
Experimental technology in hardware design	Portability
Many changes over life cycle	Flexibility Reusability Expandability
Real time application	Efficiency Reliability Correctness
On-board computer application (embedded)	Efficiency Reliability Correctness Survivability
Processing of classified information	Integrity
Interrelated systems	Interoperability

If it is difficult to pinpoint and map quality requirements for a function to the SQF's quality factors, consider surveying the software's existing or potential customers or end user's to identify what's important. Table 2-3 can be used for this purpose.

The SQF quality model identifies a set of **quality criteria**, or *software-oriented characteristics* that, if built into the software, ensure that the desired user-oriented quality factors are present. Each quality criteria *contributes* to one or more quality factors. Table 2-5 describes each of the 29 criteria and Table 2-6 shows the relationship between factors and criteria.

The selected factors and criteria, in a sense, specify high level quality requirements for the system. They identify WHAT quality characteristics the system must possess to be successful. More detailed requirements are typically needed. Goals or targets are often defined at this point to create more verifiable quality requirements. This would typically be done as part of the requirements specification process. For example, if a system must be reliable, a mean time to failure target may be identified based on an expected operational profile. Knowing how reliable the system must be will certainly influence design decisions, evaluation strategies, and software development costs.

It may be necessary to revise the results of this step once a feasibility analysis of the selected factors and criteria has been performed.

Table 2-5. Quality Criteria Definitions [SQF95]

Accuracy	Those characteristics of software which provide the required precision in calculations and output
Anomaly Management	Those characteristics of software which provide for continuity of operations under, and recovery from, non-nominal conditions
Application Independence	Those characteristics of software which determine its nondependency on database system, microcode, computer architecture, and algorithms
Augmentability	Those characteristics of software which provide for expansion of capability for functions and data
Autonomy	Those characteristics of software which determine its non-dependency on interfaces and functions
Commonality	Those characteristics of software which provide for the use of interface standards for protocols, routines, and data representations
Completeness	Those characteristics of software which provide full implementation of the functions required
Consistency	Those characteristics of software which provide for uniform design and implementation techniques and notation
Distributedness	Those characteristics of software which determine the degree to which software functions are geographically or logically separated within the system
Document Accessibility	Those characteristics of software which provide for easy access to software and selective use of its components
Effectiveness-Communication	Those characteristics of the software which provide for minimum utilization of communication resources in performing functions
Effectiveness-Processing	Those characteristics of software which provide for minimum utilization of processing resources in performing functions
Effectiveness-Storage	Those characteristics of the software which provide for minimum utilization of storage resources
Functional Overlap	Those characteristics of software which provide common functions to multiple systems

Functional Scope	Those characteristics of software which provide commonality of functions among applications
Generality	Those characteristics of software which provide breadth to the functions performed with respect to the application
Independence	Those characteristics of software which determine its non-dependency on software environment (computing system, operating system , utilities, input/output routines, libraries)
Modularity	Those characteristics of software which provide a structure of highly cohesive components with optimum coupling
Operability	Those characteristics of software which determine operations and procedures concerned with operation of software and which provide useful inputs and outputs which can be assimilated
Reconfigurability	Those characteristics of software which provide for continuity of system operation when one or more processors, storage units, or communication links fails
Self-Descriptiveness	Those characteristics of the software which provide explanation of the implementation of functions
Simplicity	Those characteristics of software which provide for definition and implementation of functions in the most noncomplex and understandable manner
System Accessibility	Those characteristics of software which provide for control and audit of access to the software and data
System Clarity	Those characteristics of software which provide for clear description of program structure in a non-complex and understandable manner
System Compatibility	Those characteristics of software which provide the hardware, software, and communication compatibility of two systems
Traceability	Those characteristics of software which provide a thread of origin from the implementation to the requirements with respect to the specified development envelope and operational environment
Training	Those characteristics of software which provide transition from current operation and provide initial familiarization
Virtuality	Those characteristics of software which present a system that does not require user knowledge of the physical, logical, or topological characteristics
Visibility	Those characteristics of software which provide status monitoring of the development and operation

Table 2-6. Relationship of Quality Factors to Criteria [SQF95]

FACTOR --> Criterion	Acronym	EFFICIENCY	INTEGRITY	RELIABILITY	SURVIVABILITY	USABILITY	CORRECTNESS	MAINTAINABILITY	VERIFIABILITY	EXPANDABILITY	FLEXIBILITY	INTEROPERABILITY	PORTABILITY	REUSABILITY
Accuracy	AC			x										
Anomaly Management	AM			x	x									
Autonomy	AU				x									
Distributedness	DI				x									
Effectiveness - Communication	EC	x												
Effectiveness - Processing	EP	x												
Effectiveness - Storage	ES	x												
Operability	OP					x								
Reconfigurability	RE				x									
System Accessibility	SS		x											
Training	TN					x								
Completeness	CP						x							
Consistency	CS						x	x						
Traceability	TC						x							
Visibility	VS							x	x					
Application Independence	AP													x
Augmentability	AT									x				
Commonality	CL											x		
Document Accessibility	DO													x
Functional Overlap	FO											x		
Functional Scope	FS													x
Generality	GE									x	x			x
Independence	ID											x	x	x
System Clarity	ST													x
System Compatibility	SY											x		
Virtuality	VR									x				
Modularity	MO				x			x	x	x	x	x	x	x
Self-Descriptiveness	SD							x	x	x	x		x	x
Simplicity	SI			x				x	x	x	x			x

Step 3 - Develop Assurance Strategy

The next step in the tailoring process involves developing a plan to validate whether or not the quality requirements defined in Step 2 are being met during the systems development or maintenance process. This plan, which describes HOW quality will be assured, can then be integrated with project plans and implemented during the project. Ideally, a life cycle approach to software quality assurance can be taken, whereby quality requirements can be checkpointed at various stages of software development, allowing the project team to take corrective action early if quality requirements are not being met with proposed designs.

Assurance of software quality can be accomplished using a number of assurance techniques. Most of the techniques involve evaluating or measuring some characteristic of either 1) an interim deliverable (i.e., a specification, software component, documentation) or 2) the end product (i.e., the operational system). Common software evaluation techniques include static reviews or inspections, and various types of software testing. Most evaluation techniques produce a measure of non-compliance.

The re-engineered SQF contains a series of inspection checklist items designed to identify whether or not a software system meets quality requirements as defined by the SQF's quality criteria. The inspection items can directly indicate whether quality requirements have been met by identifying non-compliances so that corrective action can be taken.

Strategies will be different for different functions, because quality requirements will be different. For each function and selected criterion, the strategy must specify:

- The deliverables and corresponding tasks where the criterion can be evaluated
- The evaluation technique to be used
- Resources responsible for performing the evaluation

Step 4 - Perform Feasibility Analysis

In considering the feasibility of achieving the quality requirements or goals, it is important to understand the interrelationships among the factors shown in Table 2-7. Factors with a positive relationship are complementary; achievement of high quality in one reinforces high quality in the other. Complementary relationships can also be identified by noting factors that have common criteria. Negative relationships represent conflicting requirements; achievement of high quality in one tends to lower quality in the other. Achievement of high quality in two conflicting factors is not impossible, simply more expensive. Conflicts identify the need for trade-off analysis.

Table 2-7. Factor Interrelationships [BOW85]

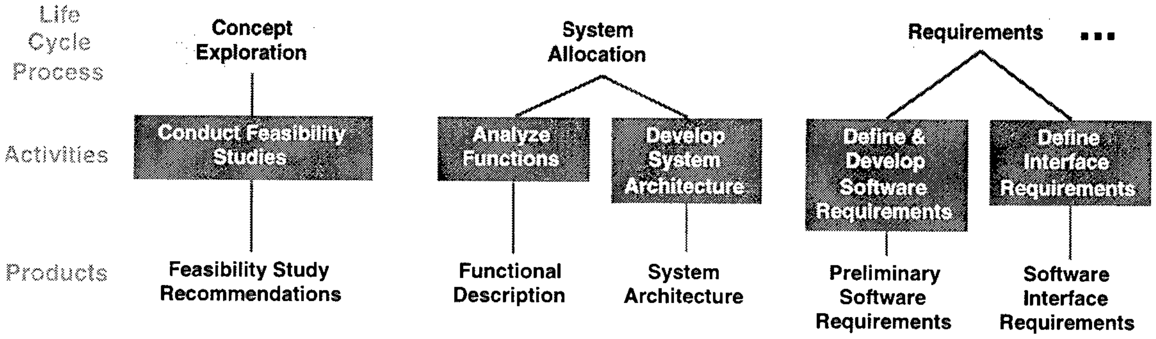
QUALITY FACTOR AFFECTED--> **QUALITY FACTOR SPECIFIED**	EFFICIENCY	INTEGRITY	RELIABILITY	SURVIVABILITY	USABILITY	CORRECTNESS	MAINTAINABILITY	VERIFIABILITY	EXPANDABILITY	FLEXIBILITY	INTEROPERABILITY	PORTABILITY	REUSABILITY
EFFICIENCY	▨						▼	▼				▼	
INTEGRITY	▼	▨											
RELIABILITY	▼		▨		▲								
SURVIVABILITY	▼	▼		▨	▲		▲		▼	▼		▼	▼
USABILITY	▼				▨		▲	▲					
CORRECTNESS						▨	▲	▲	▲	▲			▲
MAINTAINABILITY	▼						▨	▲	▲	▲			▲
VERIFIABILITY	▼							▨					
EXPANDABILITY	▼	▼	▼	▼					▨		▲		
FLEXIBILITY	▼	▼	▼	▼						▨	▲		
INTEROPERABILITY	▼	▼									▨		
PORTABILITY	▼											▨	
REUSABILITY	▼	▼	▼	▼			▲				▲	▲	▨

Positive Relationship	▲
Negative Relationship	▼

In performing trade-off analyses among the quality factors/requirements, it is necessary to consider the additional costs to build in and evaluate these quality attributes during the course of the system's life cycle. In order to estimate the cost to achieve a quality requirement, it is necessary for the cost estimation model to include an adjustment factor to account for the presence or absence of the requirement as well as the level of stringency of the requirement. We are not aware of a cost model currently in use that incorporates the 13 SQF quality factors, but believe this to be an essential step for future research.

Tailoring to Non-Waterfall Life Cycle Processes

The software development life cycle can be described in terms of the set of major processes that it contains. Each major process in turn contains a set of activities through which products are produced. Figure 2-23 shows an example of some typical life cycle processes, activities, and products. In the re-engineered SQF structure, both guidelines and indicators are linked to applicable activities and products. This linkage allows tailoring of the framework to a user's unique life cycle process.



Life Cycle Process
Concept Exploration
System Allocation
Requirements ▪▪▪
Activities
Conduct Feasibility Studies
Analyze Functions
Develop System Architecture
Define & Develop Software Requirements
Define Interface Requirements
Products
Feasibility Study Recommendations
Functional Description
System Architecture
Preliminary Software Requirements
Software Interface Requirements

Figure 2-23. Example Life Cycle Processes, Activities, and Products.

Tailoring Software Product Measurement

Products are tangible outputs of the activities that comprise the software development life cycle. Products include requirements specifications, architecture, design, code, test plans, and documentation.

Most products are revised repeatedly throughout the life cycle, during their initial creation activity and as a consequence of later rework. Prior to measurement, a product should be in a known state of completion and be under configuration management. This ensures that the measurement is both meaningful and repeatable.

One important question about product measures is:

```
Should the measurements be repeated every time the product
changes?
```

The answer depends on how the measurement will be used; specifically, on the immediacy of its consequences. Will an action be taken based on the results of the measurement? If so, it should be repeated after every significant change in the product.

For example, if there is a development standard that sets an upper limit on the control flow complexity of a procedure, then that measurement should be taken every time the procedure's implementation changes. The procedure would not be permitted to pass inspection with a value higher than the threshold.

Another reason to associate product measures with multiple versions of the product is to analyze a trend, for example, to confirm that quality is not degrading over time, or to assess the stability of a product.

Size measurements, for example, such as the total number of requirements, are used to monitor growth and stability. In this case, the analyst is looking for significant unplanned growth because it is an early indicator of effort overruns and schedule slips. This type of measurement would typically be taken at regular intervals after the end of the requirements analysis activity.

2.5.5 Guidelines for Portability

This section presents the results of the re-engineered SQF for the factor of Portability. By completing this "thread" through the framework, we have shown how the re-engineering approach could be used to redesign the entire framework. Included in this section are the following types of newly developed framework contents: Portability guidelines, Portability indicators, and Portability usage scenarios.

The quality factor Portability deals with the relative effort involved in transporting software to another environment. The IEEE dictionary [IEE91] definition is as follows:

Factor	Definition
Portability	The ease with which software can be transported from one computer system or environment to another.

In the SQF hierarchical structure, Portability decomposes into the following three criteria:

Criterion	Definition
Independence	Those characteristics of software which determine its nondependency on software environment (computing system, operating system, utilities, input/output routines, libraries).
Modularity	Those characteristics of software which provide a structure of highly cohesive components with optimum coupling.
Self-Descriptiveness	Those characteristics of software which provide explanation of the implementation of functions.

It is the criterion Independence, however, that is the essence of Portability. The other two criteria, Modularity and Self-Descriptiveness, are shared by many other factors that are also concerned with maintenance-type issues. In other words, these characteristics are beneficial in any type of future software modification undertaken by someone other than the original developer.

For this task, we limited our consideration to the criterion of Independence because this is what the published literature discusses. The guidelines in this section relate only to Independence and thus are unique to the factor of Portability.

Usage Scenarios for Portability

There are three main usage scenarios for the factor Portability.

- Development of software for multiple target environments
- Development of software that may be ported in the future
- Evaluation of the portability of existing software.

For the first two scenarios, the software is to be developed with the goal of Portability from the outset. In the first case, because the software must be delivered on multiple platforms, designing for Portability will reduce the total development effort as well as future maintenance effort. In this first scenario, the design approach is typically to maximize the amount of common code across all target environments. This reduces the testing effort as well as the implementation effort; however, the design effort may be higher to develop a common approach.

In the second case, building in Portability is an investment that is made to reduce the potential future cost of transporting the software. For this scenario, it may be more difficult to perform the trade-off analysis to determine how much to invest in building in Portability because the payoff is in the future and is unquantifiable. In this scenario, it is not likely that a great deal of design effort would be expended to reduce the proportion of system-dependent code, but rather the approach would emphasize encapsulation.

The third scenario seeks to determine the degree of portability of an existing software system, or part of a system. The reason for this evaluation may be one of the following objectives:

- Determine if transporting is feasible
- Estimate effort to transport
- Compare two or more systems or design alternatives.

This third scenario points to the need for a cost model that links the results of a Portability evaluation to the effort required to transport the software. In this scenario, the portion of the framework of interest would be those applicable to code products, to assess the "as built" system. If the requirements were rigorously maintained and documented, then it would be feasible to also review the requirements.

Portability Guidelines

We have created guidelines for the factor of Portability to provide an example of the re-engineered SQF. We have concentrated on the criterion Independence because it is the essence of Portability, and because it is also the focus of authors of reference works on the topic of Portability.

We extracted guidelines from version 1.5 of the SQF as well as from the list of references below. Table 2-8 summarizes the resulting Portability guidelines. Of the 28 guidelines, 5 were extracted from SQF 1.5, and the specific question identifier is noted in the table. Generalized guidelines were extracted from the SPC Ada 95 Quality and Style guidebook [SPC95]; guidelines dealing with Ada-specific features and implementation dependencies were omitted.

Portability References

[HAM80] Hamlet, R.G. and R.M. Haralick, "Transportable Package Software", Software -- Practice and Experience 10 (1980), pp. 1009-1027.

[HEN88] Henderson, J., Software Portability, Gower Technical Press, 1988.

[LEC86] Lecarme, O. and M.P. Gart, Software Portability, 2nd ed., McGraw-Hill, 1986.

[LIN95] Linthicum, David, "Portability Pitfalls: Include Increased Cost, Potential Dissatisfaction", Application Development Trends, May 1995.

[MOO90] Mooney, J.D., "Strategies for Supporting Application Portability", IEEE Computer , Nov. 1990, pp. 59-70.

[MOO93] Mooney, J.D., "Issues in the Specification and Measurement of Software Portability", TR 93-6, URL: http://www.cs.wvu.edu/~jdm/research/portability/reports/TR_93-6_ToC.html.

[SAX85] Saxena, S. and Field, J.A., "Portable Real-Time Software for 8-bit Microprocessors", Software -- Practice and Experience 15 (1985), pp. 227-303.

[SKA94] Skazinski, Joseph, "Porting Ada: A Report from the Field", IEEE Computer, Oct. 1994.

[SOM92] Sommerville, I., Software Engineering, 4th ed., Addison-Wesley, 1992.

[SPC95] Software Productivity Consortium, *Ada 95 Quality and Style: Guidelines for Professional Programmers*, SPC-94093-CMC, Version 01.00.10, October 1995, Chapter 7.

[WIC77] Wichmann, B. A., "Performance Considerations", Software Portability, Cambridge University Press, Cambridge, England, 1977.

Each guideline has one or more designated products and references. A designated product provides tailoring assistance and helps to link the guideline to the software development process. References provide validation of the guideline as well as a source for more detailed information.

Table 2-8. Portability Guidelines

Id	Guideline	Product(s)	Reference(s)
P.001	Use a standard subset of the implementation language; avoid non-standard extensions and obsolescent features.	Source Code	[HEN88], [LEC86], [MOO90] [SPC95] 7.1.1 [SQF95] ID.1.01,2,5
P.002	Use a version of the implementation language that is supported on other machines.	Source Code	[MOO90], [MOO93] [SQF95] ID.2.01
P.003	Select an implementation language considering the availability of translators for the anticipated target environments.	Software Design Description Source Code	[MOO93]
P.004	Build or use an automatic translator to translate from one language dialect to another.	Statement of Need Software Requirements Software Design Description	[LEC86]
P.005	Limit or isolate operations dependent on word or character size and machine dependent data element representations.	Software Design Description Source Code	[SOM92], [SPC95] 7.1.5, [SQF95] ID.2.04-5
P.006	Where data types depend on machine representations, the type definition & operations should be in a self-contained abstract data type whose underlying implementation is hidden.	Software Design Description Source Code	[SOM92]
P.007	Allow the compiler to implicitly type numeric data types.	Source Code	[LEC86]
P.008	Limit or isolate references to system-provided facilities such as libraries and utilities.	Software Design Description Source Code	[HAM80], [MOO93] [SPC95] 7.6, [SQF95] ID.1.03
P.009	Limit or isolate external input or output.	Software Architecture Software Design Description Source Code	[MOO93], [SOM92] [SQF95] ID.2.03
P.010	Eliminate all unnecessary assumptions throughout the design.	System Architecture Software Design Description	[MOO93]
P.011	For each environmental interface, either encapsulate the interface completely in a suitable module, package, object, etc., or identify a suitable standard for the interface.	System I/F Requirements Software Requirements Software Architecture Software Design Description Source Code	[MOO93], [SOM92]
P.012	Isolate operating system calls for file and process management in one or more packages.	System Architecture Software Design Description Source Code	[SOM92]
P.013	For microprocessor targets, design an intermediate level language to define low level primitives to support real-time application programming and constructs of high level languages.	System Architecture Software Design Description Source Code	[SAX85]
P.014	Design a kernel of routines that interface to the peculiar operating system of each machine, making the operating system of each machine appear identical.	Software Architecture Software Design Description Source Code	[HAM80], [SPC95] 7.6.9
P.015	Maintain a consistent structure in each environment for those elements of each interface that are used by the application.	Software Architecture Software Design Description	[MOO90]

Id	Guideline	Product(s)	Reference(s)
P.016	If a portable operating system is available, use the same operating system in all target environments.	Statement of Need Software Requirements Software Architecture	[MOO90]
P.017	Use dynamic adaptation: determine during execution the system characteristics and select appropriate methods for activities that are system-dependent.	Software Requirements Software Architecture Software Design Description Source Code	[MOO90]
P.018	In designing portable compilers, design an general intermediate language.	Software Requirements Software Architecture Software Design Description	[LEC86]
P.019	Design an intermediate level of software to form a standard interface to input/output.	Software Architecture	[LEC86]
P.020	Identify minimum necessary set of environmental requirements and assumptions.	Statement of Need Software Requirements	[MOO93]
P.021	Specifications which are sensitive to portability goals should not overly constrain the implementation.	Statement of Need Software Requirements	[MOO93]
P.022	Determine the desired degree of portability by trading off costs (such as increased development effort and/or reduced quality) and benefits (reduced effort to port to target environments).	Statement of Need Recommendations Software Requirements	[MOO93]
P.023	Correctness and ease of maintenance also enhance portability.	Software Design Description Source Code Test Summary Reported Info.	[HEN88]
P.024	Portable software should include tools for measurement and tuning of performance.	Statement of Need Software Requirements Software Design Description Source Code	[WIC77]
P.025	Use a standard character set (e.g., ASCII).	Software Requirements Software Design Description Source Code	[LEC86]
P.026	Document program portions most likely to require adaptation.	Software Design Description Source Code	[MOO90], [SPC 95] 7.1.3
P.027	Porting tasks are complicated by uncertain availability and reliability of tools used on new platforms, such as compilers, debuggers, and configuration management tools.	Statement of Need Recommendations	[SKA94]
P.028	For cross-platform development, tool selection is critical to success in building applications portable across operating systems and Graphical User Interfaces (GUIs).	Statement of Need Recommendations	[LIN95]

Evaluation of Portability

The guidelines were then broken down into inspection items and indicators to support evaluation of the factor Portability, shown in Table 2-9. Like much of the existing SQF (version 1.5) data collection forms, most guidelines are converted into questions that are answered by inspection of a product of the software development life cycle.

Our research did not uncover any quantitative metrics specifically for Portability, so the measures described in Table 2-9 were newly developed for this effort and have not been used in practice. Not all guidelines have corresponding measures. Examples of graphical representation of indicators derived from the measures are shown following the table.

Most measures are based on a count of the number of units. For small systems, one could substitute a measure of source lines of code (SLOC) in place of units.

Table 2-9. Portability Inspection Items and Measures

Id	Guideline	Inspection Items	Measures
P.001	Use a standard subset of the implementation language; avoid non-standard extensions and obsolescent features.	Is there a requirement to use a standardized implementation language? Does the unit contain any non-standard extensions of the implementation language?	# units containing non-standard extensions
P.002	Use a version of the implementation language that is supported on other machines.	Is the implementation language version supported on all target environments?	
P.003	Select an implementation language considering the availability of translators for the anticipated target environments.	Are there translators for the selected implementation language on all target environments?	
P.004	Build or use an automatic translator to translate from one language dialect to another.	Are automatic translators available to translate different language dialects?	
P.005	Limit or isolate operations dependent on word or character size and machine dependent data element representations.	Does the unit contain operations dependent on word or character size? Does the unit contain machine dependent data element representations?	# units containing word/char size dependencies # units containing machine dependent data represent.
P.006	Where data types depend on machine representations, the type definition & operations should be in a self-contained abstract data type whose underlying implementation is hidden.	Have data types dependent on machine representations been encapsulated in an abstract data type?	
P.007	Allow the compiler to implicitly type numeric data types.	Does the unit contain explicitly typed numeric data types that could be implicitly typed?	
P.008	Limit or isolate references to system-provided facilities such as libraries and utilities.	Does the unit contain a reference to a system-provided facility?	# units referencing system-provided facilities
P.009	Limit or isolate external input or output.	Does the unit perform external I/O?	# units performing external I/O
P.010	Eliminate all unnecessary assumptions throughout the design.	Are there any unnecessary assumptions in the design?	
P.011	For each environmental interface, either encapsulate the interface completely in a suitable module, package, object, etc., or identify a suitable standard for the interface.	Has each environmental interface been encapsulated? Has a standard been identified for each environmental interface?	# units containing environmental interface
P.012	Isolate operating system calls for file and process management in one or more packages.	Does the unit contain operating system calls for file or process management?	# units containing operating system calls for file or process mgt.
P.013	For microprocessor targets, design an intermediate level language to define low level primitives to support real-time application programming and constructs of high level languages.	Has an intermediate level language been defined (for microprocessor targets)?	

P.014	Design a kernel of routines that interface to the peculiar operating system of each machine, making the operating system of each machine appear identical.	Does the design provide a common interface to all targets' operating systems?	
P.015	Maintain a consistent structure in each environment for those elements of each interface that are used by the application.	Does the design provide a consistent structure for environment interfaces?	
P.016	If a portable operating system is available, use the same operating system in all target environments.	Is the operating system portable to all target environments?	
P.017	Use dynamic adaptation: determine during execution the system characteristics and select appropriate methods for activities that are system-dependent.	Has dynamic adaptation been used where feasible?	
P.018	In designing portable compilers, design an general intermediate language.	Has a general intermediate level language been designed (for compilers)?	
P.019	Design an intermediate level of software to form a standard interface to input/output.	Has the interface to I/O been standardized?	
P.020	Identify minimum necessary set of environmental requirements and assumptions.	Are there any unnecessary environmental requirements or assumptions?	# unnecessary environmental requirements
P.021	Specifications which are sensitive to portability goals should not overly constrain the implementation.	Does the portability-related specification overly constrain implementation?	
P.022	Determine the desired degree of portability by trading off costs (such as increased development effort and/or reduced quality) and benefits (reduced effort to port to target environments).	Has a portability trade-off study been performed?	
P.023	Correctness and ease of maintenance also enhance portability.		
P.024	Portable software should include tools for measurement and tuning of performance.	Is there a requirement for performance measurement and tuning tools?	
P.025	Use a standard character set (e.g., ASCII).	Is there a requirement to use a standard character set?	
P.026	Document program portions most likely to require adaptation.	Does the unit contain adequate documentation for portions likely to require adaptation?	# units inadequately documented
P.027	Porting tasks are complicated by uncertain availability and reliability of tools used on new platforms, such as compilers, debuggers, and configuration management tools.	Are reliable development tools available for all target environments?	
P.028	For cross-platform development, tool selection is critical to success in building applications portable across operating systems and Graphical User Interfaces (GUIs).	Is there a requirement to provide a consistent user interface on all target environments? Has a cross-platform development tool been selected? Is the design consistent with the capabilities provided by the selected cross-platform development toolkit?	

Portability Indicators

The following Portability indicators are derived from the measures in Table 2-9. All three can be graphically depicted in the same way, such as is shown in Figure 2-24, because all three are ratios. For all three indicators, the larger the ratio (the closer to

100%) the larger the problem, and the lower the Portability. In this example graph, the values of a computed indicator for each CSCI are compared in a column graph.

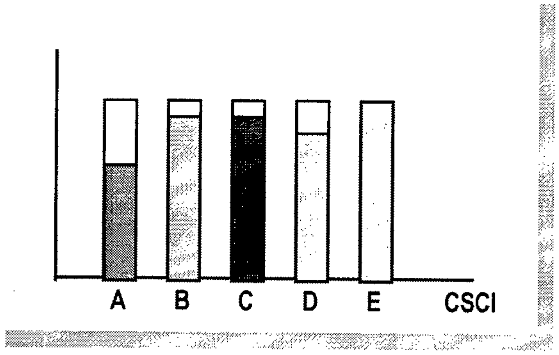


Figure 2-24. Graphing Portability Indicators

1. Over-specification - the portion of total requirements that contain unnecessary environmental requirements or assumptions.

unnecessary environmental requirements / total # requirements

2. Degree of Encapsulation - the portion of the total units that contain environmental dependencies.

(# units containing non-standard extensions + # units referencing system-provided facilities + # units performing external I/O + # units containing word/char size dependencies + # units containing machine dependent data representations + # units containing environmental interface + # units containing operating system calls for file or process mgt.) / total # units

[Note: Do not count the same unit more than once in the above equation]

3. Documentation Inadequacy - the portion of the total units that are not adequately documented for adaptation.

units inadequately documented / total # units

Deriving a Portability Inspection Checklist for Code

This section describes how the new Portability guidelines would be translated into a code inspection checklist. This is relevant to all life cycle processes that include a code inspection activity, and is relevant to all of the usage scenarios described above. The inspection could be applied during development at the completion of implementation and prior to integration testing, or it could be applied to an existing "as built" system as part of a Portability evaluation.

The first step is to identify which guidelines are applicable to the *product* "Source Code" as shown in Table 2-8. This is true for the following guidelines: P.001-P.003, P.005-P.009, P.011-P.014, P.017, and P.024-P.026. For these guidelines, extract the inspection items in Table 2-9 to create the inspection checklist for code shown in Table 2-10.

In the checklist, the guidelines are included for reference, to provide contextual information about why the question is being asked. This type of information makes it easier for the inspectors to interpret the question, and thus makes the inspections more repeatable.

Table 2-10. Portability Inspection Checklist for Code

Id	Guideline	Inspection Items	Answer
P.001	Use a standard subset of the implementation language; avoid non-standard extensions and obsolescent features.	a. Is there a requirement to use a standardized implementation language?	
		b. Does the unit contain any non-standard extensions of the implementation language?	
P.002	Use a version of the implementation language that is supported on other machines.	Is the implementation language version supported on all target environments?	
P.003	Select an implementation language considering the availability of translators for the anticipated target environments.	Are there translators for the selected implementation language on all target environments?	
P.005	Limit or isolate operations dependent on word or character size and machine dependent data element representations.	a. Does the unit contain operations dependent on word or character size?	
		b. Does the unit contain machine dependent data element representations?	
P.006	Where data types depend on machine representations, the type definition & operations should be in a self-contained abstract data type whose underlying implementation is hidden.	Have data types dependent on machine representations been encapsulated in an abstract data type?	
P.007	Allow the compiler to implicitly type numeric data types.	Does the unit contain explicitly typed numeric data types that could be implicitly typed?	
P.008	Limit or isolate references to system-provided facilities such as libraries and utilities.	Does the unit contain a reference to a system-provided facility?	
P.009	Limit or isolate external input or output.	Does the unit perform external I/O?	
P.011	For each environmental interface, either encapsulate the interface completely in a suitable module, package, object, etc., or identify a suitable standard for the interface.	a. Has each environmental interface been encapsulated?	
		b. Has a standard been identified for each environmental interface?	
P.012	Isolate operating system calls for file and process management in one or more packages.	Does the unit contain operating system calls for file or process management?	
P.013	For microprocessor targets, design an intermediate level language to define low level primitives to support real-time application programming and constructs of high level languages.	Has an intermediate level language been defined (for microprocessor targets)?	
P.014	Design a kernel of routines that interface to the peculiar operating system of each machine, making the operating system of each machine appear identical.	Does the design provide a common interface to all targets' operating systems?	
P.017	Use dynamic adaptation: determine during execution the system characteristics and select appropriate methods for activities that are system-dependent.	Has dynamic adaptation been used where feasible?	

P.024	Portable software should include tools for measurement and tuning of performance.	Is there a requirement for performance measurement and tuning tools?	
P.025	Use a standard character set (e.g., ASCII).	Is there a requirement to use a standard character set?	
P.026	Document program portions most likely to require adaptation.	Does the unit contain adequate documentation for portions likely to require adaptation?	

The next step is to tailor the checklist to the particular system or part of the system (e.g., CSCI) where it will be applied. This entails removing questions that are not applicable to any unit in the system. For example, guideline P.013 deals with microprocessor targets, so if the system does not operate on a microprocessor, the inspection item for this guideline is not applicable. Another tailoring concern is to determine which, if any, of the inspection items can be answered by using an automated tool such as a static analyzer. For Ada code, the AdaQuest version 2.2 tool (discussed in section 3.3) would provide inputs applicable to answering items P.001.b, P.005.b, and P.008.b.

Deriving Portability Measures for Code

Portability measures can be derived from Table 2-9 for the guidelines extracted for the checklist in Table 2-10. Each unit would be inspected separately, and the data would be collected from the results (i.e., the answers to the questions). The counts to be collected are summarized in the table below. The denominator of the indicator ratios should be the total number of units if an entire existing system is being evaluated for Portability. If the inspections are occurring during development, the denominator should be limited to the total number of units inspected so far, so as not to result in a falsely low value.

Indicator	Data Collected
Degree of Encapsulation	Number of units with Yes answers to any of these questions: P.001.b P.005.a P.005.b P.008 P.009 P.012
	Total number of units, or total number of units inspected
Documentation Inadequacy	Number of units with Yes answers to P.026
	Total number of units, or total number of units inspected

2.6 Support of the PSM PE Working Group

SPS strongly feels that our customer's maximum benefit from our R&D projects is achieved when our expertise and technology is applied to users in the software community. Consequently, as part of our SPS corporate philosophy, we engage in developing multiple partnering relationships that take our technology "out of the lab" and into businesses to be a force for their success. In 1994, under a separately funded project, SPS aligned itself with an on-going initiative called Practical Software Measurement (PSM), as an example of such a relationship.

The PSM initiative is funded by the Joint Logistic Commanders, Joint Group on Systems Engineering. Recently, the PSM Program Management Working Group (PSM PM WG) developed a guidebook titled *Practical Software Measurement: A Guide to Objective Program Insight* [JLC96]. The guidebook uses an issue-driven approach for tailoring measurement requirements to address the program's specific software concerns and objectives. The guidebook was designed for Program Managers who need to more objectively plan, track, and evaluate their software programs. Through the expertise of the WG participants, the guidebook is based upon proven experience from actual DoD and industry programs. With a distribution of over 3,000 copies, the guidebook has been well-received by practitioners in the software program management community. The guidebook was also endorsed by the DoD and is anticipated to be widely used.

Recently, a second PSM working group was formed to focus on Product Engineering (PE) for the software community. The objective of this newly formed Working Group is to develop guidance for ensuring quality in software products. The PSM PE guidance is envisioned to use a similar issue-driven approach as that of the PSM PM. The technology and experience base to support Product Engineering is immature compared to that of Program Management and consequently, developing a guidebook for this area is challenging.

SPS was invited to participate in this embryonic group because of our known expertise in the software quality domain and our prior successful performance on PSM PM. Alignment with this WG was beneficial to both parties because certification techniques are a vital part of product engineering. The CRC/ATD Certification Process is based upon choice of quality goals, and the PSM issue-driven approach helps the user determine how his concerns map into established quality goals. We felt our support was an excellent opportunity to secure valuable feedback from the user community with respect to our research and development on CRC/ATD. In an effort to transfer CRC/ATD-developed technology to industry users, SPS felt that it was mandatory to align ourselves with members of the software community who could apply our research. The results of the meeting discussions of August 20-21, 1996, October 30, 1996, and December 18, 1996 appear in Appendix A.

Some of the participants of the PSM PE WG were also participants in the PSM PM WG; however, other participants are new to the initiative. The participants of the PSM PE WG represent the following organizations:

- Office of the Under Secretary of Defense (OSD A&T)
- SPAWARSYSCOM
- Defense Information Systems Agency (DISA)
- National Security Agency (NSA)
- Rome Laboratory
- Navy CU-UDE
- Naval Center for Cost Analysis
- Naval Undersea Warfare Center (NUWC)
- U.S. Army MICOM
- U.S. Army Materiel Command
- National Park Service
- SBA
- Lockheed Martin
- TRW
- Hughes
- Logicon
- Software Productivity Solutions, Inc.
- Independent Engineering
- CERC
- MITRE
- Institute for Defense Analysis (IDA)
- Virginia Polytechnic Institute (VPI)

As seen in the results of the meetings, the scope of the Product Engineering WG was established and an initial identification of the issue was accomplished. SPS proceeded to categorize the issues into a organization that was patterned after the PSM PM guidebook. The results of that activity appears in Table 2-11.

Table 2-11 SPS' Attempt at PSM PE Issue Categorization

Common Issue	Attribute/Category
Defined Capabilities (Is it right functionally and structurally?)	**Completeness** (all expected functionality present?) **Correctness** (What the user intended?) **Consistency** **Understandability** (of Documentation/ Clarity) **Conformance to standards** **Precision** (Computational/Accuracy)
Operational Performance (Will it operate adequately?)	**Reliability** **Safety** **Security** **Efficiency** **Throughput** (real-time Performance) **Fault Tolerance** (Redundancy/ Survivability) **Availability** **Usability** (Ease of Use)
Adaptability (Will it meet future needs?)	**Expandability** (How easily can functionality be added?) **Maintainability** **Scalability** **Flexibility** **Portability** (Common Op. Environ.)
Development Effort (How hard will it be to build?)	**Traceability** (Quality Flow-Through) **Installability** **Testability**

The above issue organization was presented at the PSM PE WG meeting and was well received by its participants. However, members indicated that "ilities" were difficult to measure objectively and preferred to minimize their use.

The following additional notes discuss the remaining issues proposed by the group that do not appear in the categories. For a complete list of all issues proposed by the group, See Appendix A, minutes of 6 November 1996 meeting. For example, "Stability of Requirements" and "Requirements Volatility" were deemed out of scope, because these relate more to program management than product engineering. Items such as risks and objectives should be covered in the guidance; but they are not part of the framework. The Correctness issue is primarily measured by non-correctness (i.e., defects). Therefore proposed issues such as "Conflicting Requirements" should be considered categories of the measure, because they are types of defects. Supportability was deleted because it was seen as Expandability (enhance) and Maintainability (fix).

The following items did not appear to be product issues and should not be part of the Product Engineering framework:

- User Satisfaction
- Defect of Quality
- Defect of Quality Requirements
- Predictability or Attainment of Quality Objectives
- Acceptance Criteria
- Concurrent Engineering
- Defect Recognition (When inserted, found, fixed)
- Defect of Failure
- Prediction of Defects
- Readiness to Deliver

The following proposed items are measures, not issues or categories. They are in the scope of the framework, but shouldn't be addressed at the level of issues:

- MTTR
- Product Size
- Complexity
- Standard Product Quality Ranges

SPS anticipates continuing our support of the PSM PE WG under a separately funded effort.

3.0 SQF Application to Reuse Certification

The objective of this task was to extract useful techniques from the SQF to be applied to reuse certification. The first step was to analyze the SQF to determine potential contributions to reuse certification, and this step is discussed in section 3.1. Of these potential contributions, three were executed as tasks under this effort, and the results are documented in sections 3.2 through 3.4. The fourth contribution influenced the defect prediction model developed by the MITRE corporation under a separate but related contract. The results of the SQF contribution to the defect prediction model is discussed in section 3.5.

3.1 Analysis of the SQF for Reuse Certification

The SQF was analyzed to determine how the quality assessment methodology, techniques, and metrics embodied within the SQF could be valuable in the certification of reusable assets. Four potential contributions were identified and are discussed in more detail in the following subsections.

- Code inspection checklist
- Automated static analysis
- Guidance for building quality factors into reusable code assets
- Predictive model of quality

3.1.1 SQF as a Code Inspection Checklist

Code review through inspection is a static quality evaluation technique on the list of CRC "best bet" techniques because of its effectiveness at finding defects. There is much written about code inspections, and there are many inspection checklists in the literature, but none are specific to reuse certification needs. Since code inspection is a significant part of the SQF methodology, the SQF data collection forms are a potential resource for developing a reuse certification-specific checklist.

3.1.2 Automated Static Analysis

The SQF incorporates many measurements which could be obtained through automated static analysis of code. There is a clear cost benefit to collecting measures with an automated tool when possible as opposed to inspection by a human. The RC-SQF effort demonstrated automated measurement with the AdaQuest static analysis tool, and identified additional targets for automation such as style guideline checks.

The certification process for code assets developed under the CRC effort incorporates a static analysis step chiefly because of the cost-effectiveness of this technique. To be useful in the context of certification, a tool which supports the static analysis step must clearly show defects or violations in the code.

3.1.3 Guidance for Building Quality Factors into Reusable Code Assets

With up-front guidance on how to achieve quality, with a "quality blueprint", developers of reusable assets will generate more assets that pass certification for applications that require high levels of quality. This guidance can be extracted from the SQF using the re-engineering approach described in section 2.4. Since the Certification Framework and certification process developed under the CRC effort concentrates on the certification concerns of Correctness, Completeness and Understandability, these concerns are targets for extraction of guidance from SQF. The certification concern of Correctness maps directly to the SQF factor of the same name.

3.1.4 Predictive Quality Model

The certification process developed on the CRC effort concentrates on the reuse certification quality concerns of Correctness, Understandability, and Completeness, and therefore the certification techniques focus on finding these types of defects. The Certification Framework guides the certifier to select the most cost-effective certification techniques based on an historical profile of expected defects. An alternative to an historical profile of defects would be a prediction of defects by measuring key indicators in the code sample to be certified. This predictive model would, ideally, be computed via automated static analysis of code and could predict defects by defect classification categories (data, logic, interface, etc.).

The SQF approach combines measures from data collection forms to predict aspects of the quality of the end product—a software system or CSCI. For an inspection technique, a checklist should be comprehensive, so you don't miss anything. A predictive model, on the other hand, is meant to be concise, so that you only measure the indicators that are strongly correlated to the aspect you are trying to predict. The model should also be comprised of independent indicators, so that you are not measuring essentially the same effect in several different ways when one will suffice.

The MITRE corporation has developed a defect prediction model under a separate but related contract with Rome Laboratory. The initial version of the model predicts the total number of defects expected to be found in a unit of code based upon characteristics of the code as well as the development environment. The key model parameters are established empirically with data from the NASA Software Engineering Laboratory (SEL). In order to be more useful for the certification of reusable components, we would like the model to be able to predict defects by type categories according to the CRC code defect model. These categories are Computational, Data, Interface, Logic and Other.

The SQF is rich in measures of different aspects of code complexity as illustrated in Figure 3-1. It is hypothesized that complexity measures could be added to the MITRE model to attempt to predict the number of defects by category. The basic idea is that complex code is more likely to contain defects; therefore, for example, code with high data complexity is more likely to contain data defects.

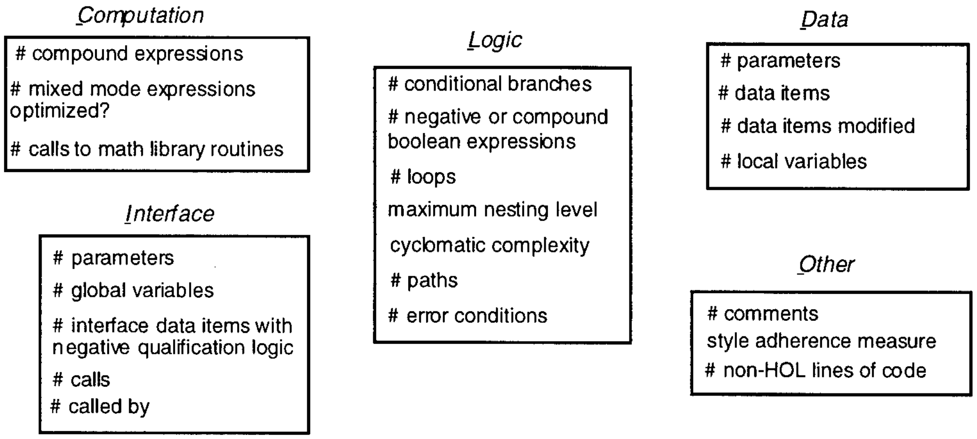


Figure 3-1. Using Complexity Measures to Predict Defect Types.

3.2 Reuse Certification Code Checklists for Ada and C++ Assets

Two code inspection checklists were developed under this effort. The first, for Ada code, was used in the first certification field trial, performed under the CRC effort and documented in the CRC Volume 5 Certification Field Trial. The Ada checklist is discussed in section 3.2.1. The second checklist, for C++ code, was used in the second certification field trial performed under this effort (results summarized in section 4.1.1). The C++ checklist is discussed in section 3.2.2.

3.2.1 Ada Code Inspection Checklist

Figure 3-2 illustrates how the Ada code inspection checklist was constructed.

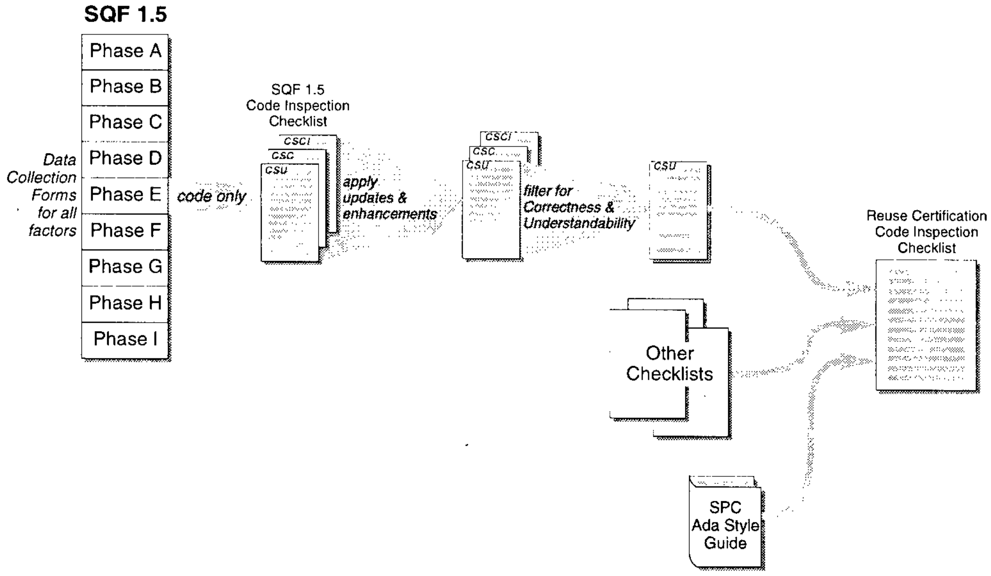

SQF 1.5
Phase A
Phase B
Phase C
Phase D
Phase E
Phase F
Phase G
Phase H
Phase I
Data Collection Forms for all factors
code only
SQF 1.5 Code Inspection Checklist
CSCI
CSC
CSU
apply updates & enhancements
CSCI
CSC
CSU
filter for Correctness & Understandability
CSU
Reuse Certification Code Inspection Checklist
Other Checklists
SPC Ada Style Guide

Figure 3-2. Method for Constructing Ada Code Inspection Checklist

The Ada code checklist was customized for the certification process in two ways. First, each inspection checklist item was pre-categorized as belonging to either the Correctness or Understandability concern and to a specific defect category. Each item was given an identifier to encode this categorization to simplify field trial data analysis. The second customization was that checklist items that were detected by the automated static analysis tools used in the second step of the certification process were removed from the checklist. Thus there was no intentional duplication between these two steps of certification process. It must be noted that selection of questions to remove is dependent on the capabilities provided by the automated tools used for static analysis. For another set of tools, the result would not necessarily be the same.

The last column in Table 3-1 is the source, or reference, for the checklist item. As Figure 3-2 shows, the checklist was derived from multiple sources including the SQF. All source checklists are collated in Appendix B for the reader's convenience, and are listed in tabular form. A check mark indicates which items were selected. For rejected questions, a comment explains why that question was rejected for the reuse-specific checklist.

The resulting checklist for Ada code is given in Table 3-1. The first column of the table is the unique identifier for each checklist item. The identifier is coded to indicate the defect category and certification concern. The first letter indicates the defect type (C = Computational, D = Data, I = Interface, L = Logic, and O = Other). The last letter suffix indicates the certification concern (C = Correctness, and U = Understandability).

A sample checklist blank data collection form is included in Appendix B. This sample checklist is identical to the one used in the first CRC Field Trial.

Table 3-1. Code Inspection Checklist for Ada Code

Identifier	Question	Source
	• Computational •	
C.01.U	For functions that perform computations, are accuracy tolerances documented?	[SQF95] AC.1.5.e
C.02.C	Do all computations use variables with consistent types, modes, and lengths? (e.g., no Boolean variables in arithmetic expressions, or mixed integer and floating-point)?	[EBE94], [SQF95] EP.2.5.e, [NAS94]
C.03.C	Are all expressions free from the possibility of an underflow or overflow exception?	[EBE94]
C.04.C	Are all expressions free from the possibility of a division by zero?	[EBE94], [DUN84]
C.05.C	Is the order of computation and precedence of operators correct in all expressions?	[EBE94]
C.06.C	Are all expressions free from invalid uses of integer arithmetic, particularly divisions?	[EBE94]
C.07.C	Are all computations free from nonarithmetic variables?	[EBE94]
C.08.C	Are all comparisons between variables of compatible data types, modes, and lengths?	[EBE94]
C.09.C	Do all comparisons avoid equality comparison of floating-point variables?	[EBE94], [NAS94]
C.10.C	Is the code free from assignment of a real expression to an integer variable?	[NAS94]
C.11.C	Are all bit manipulations correct?	[NAS94]
	• Data •	
D.01.C	Are all data items referenced?	[SQF95] CP.1.6.e, [DUN84]
D.02.U	Do all references to the same data use single unique names?	[SQF95] CS.2.14.e
D.03.C	Are all character strings complete and correct, including delimiters?	[FAG76]
D.04.C	Are illegal input values systematically handled?	[ONE88], [EBE94], [DUN84]
D.05.C	Are all variables set or initialized before referenced?	[EBE94]
D.06.C	Are all array indexes integers?	[EBE94]
D.07.C	For all references through pointer variables, is the referenced storage currently allocated?	[EBE94]

D.08.C	Are all storage areas free from alias names with different pointer variables?	[EBE94]
D.09.C	Are all variables correctly initialized?	[EBE94], [BEA94]
D.10.C	Are all variables assigned to the correct length, type, storage class and range?	[EBE94], [BEA94]
D.11.U	Is the code free from variables with similar names (e.g., VOLT and VOLTS)	[EBE94]
D.12.C	Are all indexes properly initialized?	[DUN84]
D.13.U	Are all data declarations commented?	[BEA94]
D.14.U	Are all data names descriptive enough?	[BEA94]
D.15.C	Are constant values used only as constants and not as variables?	[NAS94]
D.16.C	For all arrays, is the attribute ‘RANGE used instead of numeric literals?	[SPC95]
D.17.U	Are error tolerances documented for all external input data?	[SQF95] AM.2.1.e

• Interface •

I.01.C	Are all propagated exceptions declared as visible and documented?	[SQF95] AM.1.5.e
I.02.C	Are all propagated exceptions handled (not raised) by the calling unit?	[SQF95] AM.1.5.e, [BEA94]
I.03.C	Are reasonable ranges declared for all output values?	[SQF95] AM.3.4.e
I.04.C	For all global variables, is their use justified, and are they documented?	[SQF95] AP.2.2.e, AP.2.3.e
I.05.U	Are all subprogram parameters modes shown and usage described via comments?	[SQF95] AP.2.4.e, CP.1.3.e, [NAS94]
I.06.U	Does the prologue document all side effects, such as propagated exceptions?	[SQF95] SD.2.1.e
I.07.U	Are there any interface data items with negative qualification logic (e.g., Boolean values that return "true" upon failure rather than success)?	[SQF95] ST.1.3.e
I.08.C	Do all units systems of formal parameters match actual parameters (such as degrees vs. radians, or miles per hour vs. feet per second)?	[EBE94], [NAS94]
I.09.C	Are all functions free from modification of input parameters?	[EBE94]
I.10.C	Are global variables consistently used in all references?	[EBE94]
I.11.C	Are files opened before use?	[EBE94]

I.12.C	Are all input parameter variables referenced? Are all output values assigned?	[DUN84], [SQF95] CP.1.11.e, [NAS94]
I.13.U	Does each unit have a single function, and is it clearly described?	[NAS94]
I.14.C	Are all functions free from side effects?	[SPC95]
I.15.C	Is there a single entry and a single exit?	[ONE88]

Selection Criteria

The criteria for selecting items for the reuse-specific checklist are as follows:

- Must address reuse certification concerns of Correctness, Understandability, or Completeness.
- Must be applicable to code assets (as opposed to design, or test case)
- Must not require knowledge about the development process used to create the asset
- Must be applicable to Ada, C or C++
- Must be a Yes/No question
- Must not be subjective

In fact, the reuse certification concern of Completeness was not addressed in the checklist because of the way the field trial procedures were designed. In the field trial procedures, Completeness was addressed up front in the Readiness step, and not in the Code Inspection step.

SQF as a Source

SQF selections were limited to CSU-level questions for *code* products. The data collection form for the Code and Unit Test Phase (i.e., Phase E) from [SQF95] was the starting point. The Phase E DCF includes questions at the CSC level and above, and also includes questions that pertain to documentation, problem reports, and test plans. The CSU-level code questions from Phase E DCF are shown as the SQF source in Appendix B.

SQF selections were not limited to the questions for the SQF factor of Correctness. All factors were included for consideration. Many SQF questions were edited, for example, to convert from numeric ratios to Yes/No questions, to make the questions more Ada-specific, or to combine related questions. The Ada Style and Quality Guidelines [SPC91] were consulted in cases where an Ada-specific interpretation was made. Figures 3-3, 3-4, and 3-5 illustrate some examples of how SQF questions were modified.

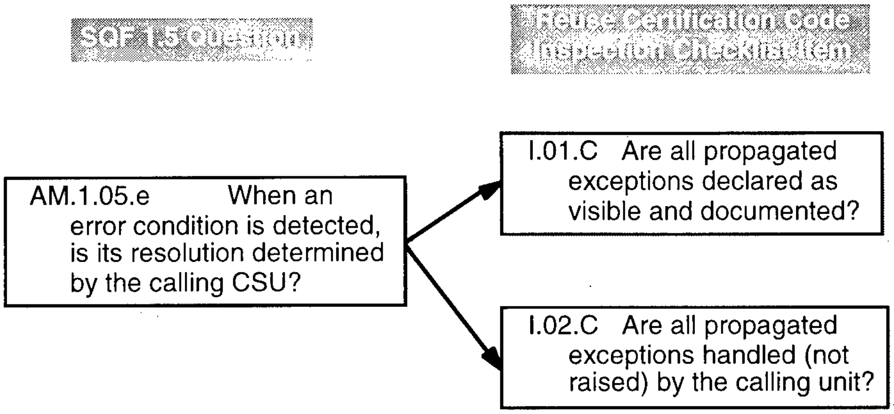

SQF 1.5 Question
Reuse Certification Code Inspection Checklist Item
AM.1.05.e When an error condition is detected, is its resolution determined by the calling CSU?
I.01.C Are all propagated exceptions declared as visible and documented?
I.02.C Are all propagated exceptions handled (not raised) by the calling unit?

Figure 3-3. Converting an SQF Question to a More Ada-Specific Interpretation.

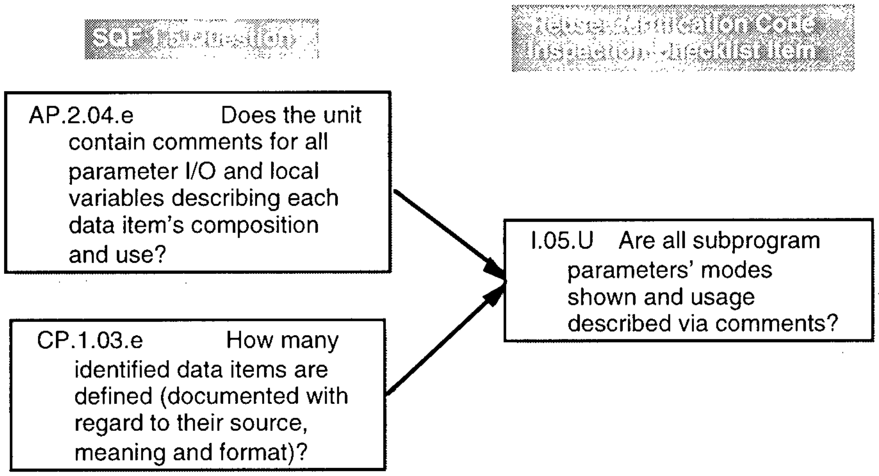

SQF 1.5 Question
Reuse Certification Code Inspection Checklist Item
AP.2.04.e Does the unit contain comments for all parameter I/O and local variables describing each data item's composition and use?
CP.1.03.e How many identified data items are defined (documented with regard to their source, meaning and format)?
I.05.U Are all subprogram parameters' modes shown and usage described via comments?

Figure 3-4. Combining Related SQF Questions.

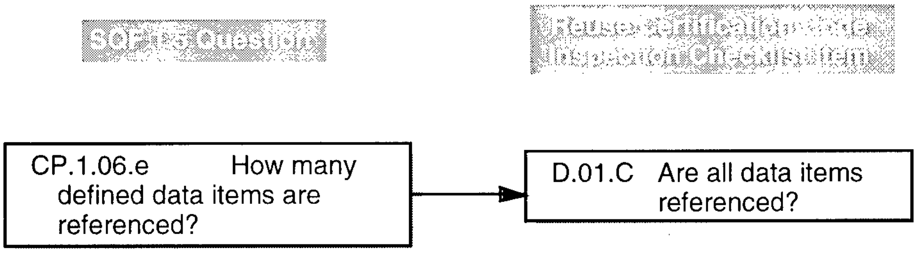

SQF 1.5 Question
Reuse Certification Code Inspection Checklist Item
CP.1.06.e How many defined data items are referenced?
D.01.C Are all data items referenced?

Figure 3-5. Converting an SQF Ratio Question to a Yes/No Question.

3.2.2 C++ Code Inspection Checklist

Figure 3-6 illustrates how the C++ code inspection checklist was constructed.

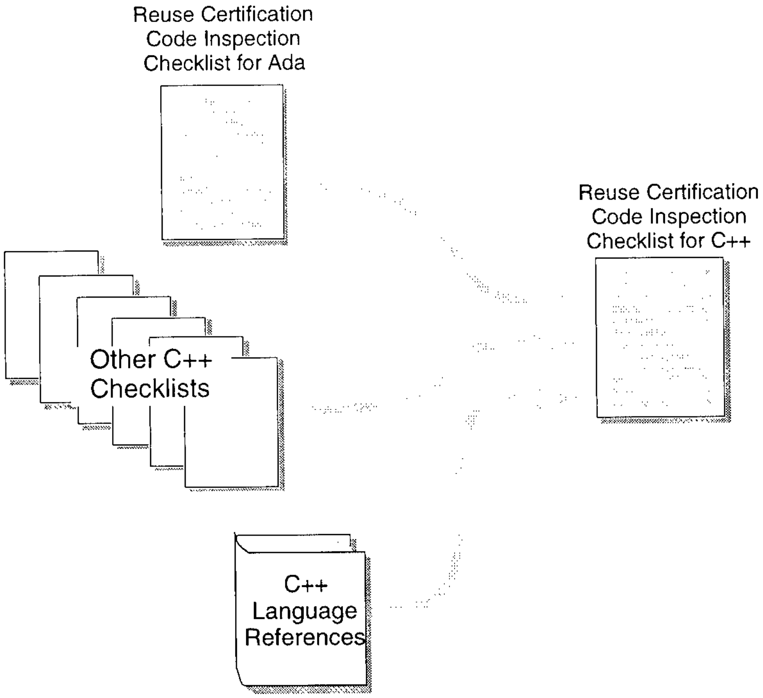

Reuse Certification Code Inspection Checklist for Ada
Reuse Certification Code Inspection Checklist for C++
Other C++ Checklists
C++ Language References

Figure 3-6. C++ Code Inspection Checklist

Initially, we thought that adopting an existing checklist would be the best solution to acquiring a code checklist for C++. We surveyed industry literature and examined six existing C++ Code Checklists [BAL92], [DST96], [FAG96], [FAU94], [GER95], [HUM95], [KOE92], [KOE95], [MCC96], [POT94], [SOF95], [SOF96], and [VAN95]. We found that no one checklist was appropriate. Each had goals different than reuse and certification; each used different levels of granularity.

Therefore, we chose to modify the existing Reuse Certification Code Inspection Checklist for Ada developed under CRC in light of this checklist research. This approach resulted in a historical flowdown from the Ada checklist to the C++ checklist, and provided continuity with limits on the variability. Our approach also provided additional detail to the existing checklist, with the goal of improving its effectiveness. Checklist questions were either modified, added, or deleted.

A summary of modifications of the Ada code checklist for C++ appears in Table 3-2 and Figure 3-7.

Table 3-2. Modifications of Code Checklist

		Questions		
Defect Class	Unchanged	Modified	Added	Deleted
Computational	6	3	5	0
Data	9	6	25	1
Interface	9	3	15	2
Logic	15	2	16	0
Other	4	1	2	0

The data is better viewed in graph form as shown in Figure 3-7.

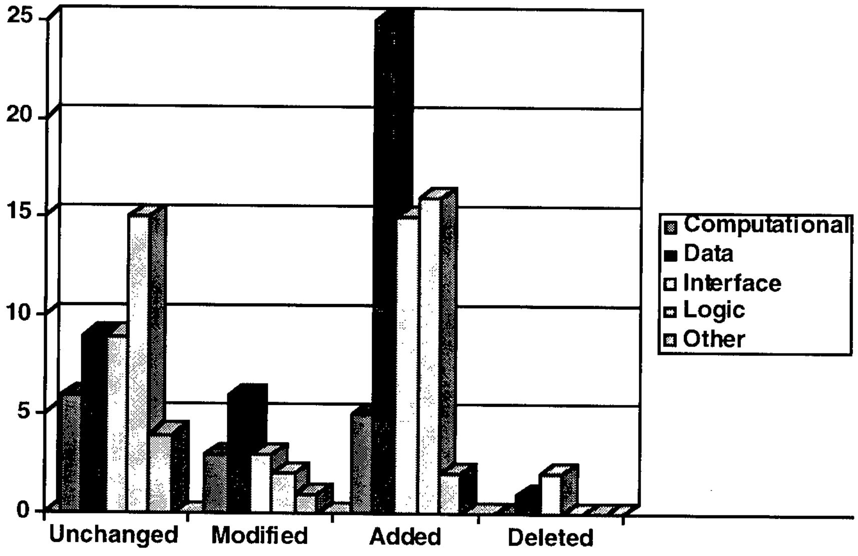


Figure 3-7. Modifications to the Code Checklist

The results of the reuse certification code inspection checklist for C++ appear in Table 3-3. For the second Field Trial with a C++ code asset, formatting conventions were observed to document updates and refinements to the code checklist as an attempt to preserve the integrity of the checklist from the first Field Trail with an Ada code asset. Specifically, italicized checklist questions indicate those questions used for Ada source code in the first Field Trial. Non-italic questions are those that have been added to modify the existing Ada checklist for a C++ source code component in the second Field Trial. Strikethroughs in the checklist questions indicate that the question was valid for an Ada source code component, but was not appropriate for a C++ source code component due to specific language characteristics.

Table 3-3. Code Inspection Checklist for C++ Code

Identifier	*Question*	Source	Guideline
	• Computational •		
C.01.U	*For functions that perform computations, are accuracy tolerances documented?* For functions that perform computations, are accuracy tolerances documented for variable types that hold data?	[SQF95] AC.1.5.e	
C.02.C	*Do all computations use variables with consistent types, modes, and lengths (e.g., no boolean variables in arithmetic expressions, or mixed integer and floating-point)?* Do all computations use variables with consistent types and/or type casting, values, and lengths? (i.e., no boolean variables in arithmetic expressions) If variable types are mixed, are expected outcomes anticipated and external to the program block?	[EBE94] [SQF95] EP.2.5.e [POT94] p. 139	
C.03.C	*Are all expressions free from the possibility of an underflow or overflow exception?*	[EBE94]	
C.04.C	*Are all expressions free from the possibility of a division by zero?*	[EBE94]	
C.05.C	*Is the order of computation and precedence of operators correct in all expressions?*	[ENE94] [POT94] p. 128	
C.06.C	*Are all expressions free from invalid uses of integer arithmetic, particularly divisions?*	[EBE94]	
C.07.C	*Are all computations free from non arithmetic variables?*	[EBE94]	
C.08.C	*Are all comparisons between variables of compatible data types, modes, and lengths?* Are all comparisons between variables of compatible data types, type cast data types, and lengths?	[EBE94] [POT94] p. 139	
C.09.C	Do all comparisons avoid equality comparison of floating-point variables?	[EBE94]	[SPC91] 5.5.6.a
C.10.C	Is the code free from assignment of a real expression to an integer variable?	[NAS94]	
C.11.C	*Are all bit manipulations correct?*	[NAS94]	
C.12.C	Is the "%" modulus operator used correctly (i.e. not intended as a percentage)?	[POT94] p. 126	
C.13.C	Is the "/" division operator used to accommodate a discarded remainder?	[POT94] p. 127	

Identifier	*Question*	Source	Guideline
C.14.C	Are compound operators assigned correctly?	[POT94] p. 136	

• Data •

D.01.C	*Are all data items referenced?*	[SQF95] CP.1.6.e	
D.02.U	*Do all references to the same data use single unique names?*	[SQF95] CS.2.1.4.e	
D.03.C	*Are all character strings complete and correct, including delimiters?* Are all character strings and character arrays complete and correct, including delimiters (i.e., value is assigned and enough elements are reserved to hold entire character string and terminating null zero)?	[FAG76] [POT94] p. 56, 76, 78	
D.04.C	*Are illegal input values systematically handled?*	[ONE88] [EBE94]	
D.05.C	*Are all variables set or initialized before referenced?*	[EBE94]	[SPC91] 5.9.6.a
D.06.C	*Are all array indexes integers?*	[EBE94]	
D.07.C	*For all references through pointer variables, is the referenced storage currently allocated?*	[EBE94]	[SPC91] 5.4.3.d
~~D.08.C~~	~~*Are all storage areas free from alias names with different pointer variables?*~~	~~[EBE94]~~	~~[SPC91] 5.4.3.e~~
D.09.C	*Are all variables correctly initialized?* Are all variable and constants correctly initialized?	[EBE94]	5.4.3.e [SPC91]
D.10.C	*Are all variables assigned to the correct length, type, storage class and range?* Are all variables and constants assigned to the correct length, type, sign, precision, and range?	[EBE94] [POT94] p. 58	
D.11.U	*Is the code free from variables with similar names (e.g., VOLT and VOLTS)?* Is the code free from variables and constants with similar names (e.g., VOLT and VOLTS)?	[EBE94] [POT94]	
D.12.C	*Are all indexes properly initialized?* Are all indexes properly initialized (i.e., start at zero)?	[DUN84] [POT94] p.79	
D.13.U	*Are all data declarations commented?*	[BEA94]	
D.14.U	*Are all data names descriptive enough?*	[BEA94]	
D.15.C	*Are constant values declared as constants and not as variables?* Are constant values used as numbers, characters, words, or phrases?	[NAS94] [POT94] p.46	[SPC] 3.2.5.b.e

Identifier	*Question*	Source	Guideline
D.16.C	For all arrays or enumeration types, are ranges used for each data type instead of numeric literals?	[SPC91]	[SPC] 5.5.2.b,c
D.17.U	*Are error tolerances documented for all external input data?*	[SQF95] AM.2.1.e	
D.18.U	Are variable names in lower case as is the customary convention?	[POT94] p. 53	
D.19.U	For object-oriented code, are the first letters of class names capitalized as is the customary convention?	[POT94] p. 53	
D.20.U	Are upper case letters used for "#define" directives as is the customary convention?	[POT 94] p. 91	
D.21.U	Are "#define" statement used judiciously?	[POT94] p. 96	
D.22.C	Are assignment equals "=" and equals to "==" operators used correctly?	[POT94] p. 144	
D.23.C	Have assignment expressions been included in the same condition as the logical test?	[POT94] p. 165	
D.24.U	Are parenthesis used in the expressions of the "sizeof" operator (i.e., in "sizeof data", parentheses is optional, but it is good programming to include (); Are parenthesis used in the expressions of the "sizeof (data type) where parentheses are required?	[POT94] p. 181	
D.25.C	Are bitwise operators, bitwise shift, and compound bitwise shift used correctly (i.e., &, vertical bar, ^, ~, >>, <<, <<=, >>=)?	[POT94] p. 186	
D.26.C	For object-oriented components, do classes have any virtual functions? If so, is the destructor non-virtual?	[BAL92] p. 5	
D.27.C	For object-oriented components, do classes have all three necessary copy-constructors, assignment operators, and destructors?	[BAL92] p. 5	
D.28.C	For object-oriented components, do all structures and classes use the "." reference?	[HUM95] [GER95]	
D.29.C	Are all pointers initialized to "null", deleted only after "new", and new pointers deleted after use?	[HUM95]	
D.30.C	Are names used within the declared scope?	[GER95]	
D.31.C	For object-oriented components, is each class declared and implemented in a single file (i.e., with the exception of helper classes packaged with the primary file)?	[VAN95]	

Identifier	*Question*	Source	Guideline
D.32.C	Are function arguments free from variable argument lists (...) to avoid the inherently type-unsafe?	[VAN95] rule 6.3	
D.33.U	Is multiple inheritance avoided?	[VAN95] rule 12.2	
D.34.U	Are "return" types always provided, even if "void"?	[DST96]	
D.35.C	For object-oriented components, does every constructor initialize every data member in its class?	[KOE92] p. 44	
D.36.C	For object-oriented components, do assignment operators correctly handle assigning an object to itself?	[KOE92] p. 44	
D.37.C	Is "delete [ ]" used when deleting an array to determine the size of the array being deleted?	[KOE92] p. 43	
D.38.U	For object-oriented components, are object fine grained?	[MCC96] p. 7, 10	
D.39.U	For object-oriented components, is the object encapsulated (i.e., highly related methods and data isolated)?	[MCC96] p. 7, 10	
D.40.U	For object-oriented components, is there low dependency between objects?	[MCC96] p. 7, 10	
D.41.U	For object-oriented components, do objects exhibit high fan in?	[MCC96] p. 8	

• Interface •

~~I.01.C~~	~~*Are all propagated exceptions declared as visible and documented?*~~	~~[SQF95] AM.1.5.e~~	~~4.3.1.d,e~~
~~I.02.C~~	~~*Are all propagated exceptions handled (not raised) by the calling unit?*~~	~~]SQF95] AM.1.5.e~~	~~4.3.1.1~~ ~~5.8.3.a,b~~
I.03.C	*Are reasonable ranges declared for all output values?*	[SQF95] AM.3.4.e	
I.04.C	*For all global variables, is their use justified, and are they documented?*	[SQF95] AP.2.2.e, AP.2.3.e	4.1.6.a, 4.2.1.f
I.05.U	*Are all subprogram parameter modes shown and usage described via comments?* Are all subprogram parameter types shown and usage described via comments?	[SQF95] AP.2.4.e, CP.1.3.e, [NAS94]	5.2.4.a
I.06.U	*Does the prologue document all side effects, such as propagated exceptions?* Does the prologue document all side effects?	[SQF95] SD.2.1.e	
I.07.U	*Are the interface data items free from negative qualification logic (e.g., boolean values that return "true" upon failure rather than success)?*	[SQF95] ST.1.3.e	5.5.4.a,b

Identifier	*Question*	Source	Guideline
I.08.C	*Do all units systems of formal parameters match actual parameters (such as degrees vs. radians, or miles per hour vs. feet per second)?*	[EBE94]	
I.09.C	Are all functions free from modification of input parameters?	[EBE94]	
I.10.C	*Are global variables consistently used in all references?*	[EBE94]	
I.11.C	*Are files opened before use and closed when finished?* Are files opened immediately prior to access and closed as soon as done?	[EBE94] [POT94] p. 502	
I.12.C	*Are all input parameter variables referenced? Are all output values assigned?*	[DUN84] [SQF95] CP.1.11.e [NAS94]	
I.13.U	*Does each unit have a single function, and is it clearly described?*	[NAS94]	
I.14.C	*Are all functions free from side effects?*	[SPC91]	4.1.3.b
I.15.C	*Is there a single entry and a single exit?*	[ONE88]	
I.16.C	Does the program and all its functions end with a return statement?	[POT94] p. 48, 283	
I.17.C	Does each return have a closing brace (i.e., after the end of a block, the end of the main function [main ( )], and the end of the program?	[POT94] p. 48	
I.18.C	Are the widths and formats of numbers specified correctly for printing?	[POT94] p. 108	
I.19.C	Are the most frequently executed statements in a "switch" arranged at the top of the list to improve the efficiency of the code?	[POT94] p. 260	
I.20.C	If "ios::out" is used to open a file for writing (i.e., C++ creates the file), does it overwrite the filename that exists?	[POT94] p. 501	
I.21.U	Is code free from "non-standard" syntactic constructs such as unconventional preprocessor directives?	[FAU94]	
I.22.C	Is passing objects by value, or by reference avoided (e.g., where implicit conversions result in member wise copying)? Are dynamically allocated application objects passed as pointers?	[FAU94]	
I.23.C	To decrease performance overhead, are local variables created and assigned at once?	[BAL92] p. 6	

Identifier	*Question*	Source	Guideline
I.24.C	Are files properly declared, opened, and closed?	[HUM95] [GER95]	
I.25.C	Is a file closed in the case of an error return?	[BAL92] p. 16{	
I.26.C	Are all "include" statements complete?	[HUM95] [POT94]	
I.27.C	Are "inline" functions used only when performance is needed?	[VAN95]	
I.28.C	Are "new" and "delete" used to allocate and deallocate storage rather then "malloc" and "free" (i.e., which are type-unsafe)?	[VAN95] [BAL92]	
I.29.C	Have timing, sizing, and throughput been addressed?	[BEA94] p. 8	

• Logic •

L.01.C	*Are all negative boolean and compound boolean expressions correct?*	[SQF95] SI.4.3.e	
L.02.C	*For all case statements, is the domain partitioned exclusively and exhaustively?* For all "switch" statements, is the domain partitioned exclusively and exhaustively?	[ONE88] [POT94]	
L.03.C	*Are all indexing operations and subscript references free from off-by-one defects?*	[EBE94]	
L.04.C	*Are all comparison operators correct?*	[EBE94]	
L.05.C	*Are all boolean expressions correct?*	[EBE94]	
L.06.C	*Is the precedence or evaluation order of boolean expressions correct?*	[EBE94]	
L.07.C	Do the operands of boolean expressions have logical values (0 or 1) or a non zero value which is interpreted as true?	[EBE94] [POT94] p. 47	
L.08.C	*Does every loop eventually terminate?*	[EBE94] [ONE88]	
L.09.C	*Is the program free from goto statements?* Are "gotos" used judiciously or can other code be substituted?	[EBE94] [POT94] p. 271	5.6.7.a
L.10.C	*Are all loops free from off-by-one defects (i.e., more than one or fewer than one iteration)?*	[EBE94]	
L.11.C	Are all switch statements free from "others" branches?	[SPC91]	5.6.3
L.12.C	*Are all decisions exhaustive?*	[EBE94]	
L.13.C	*Are end-of-file conditions detected and handled correctly?*	[EBE94]	

Identifier	*Question*	Source	Guideline
L.14.C	*Are end-of-line conditions detected and handled correctly?*	[EBE94]	
L.15.C	*Do processes occur in the correct sequence?*	[DUN94]	
L.16.C	*Are all loops free from unnecessary statements?*	[DUN84]	
L.17.C	*Are all loop limits correct?*	[BEA94]	
L.18.C	*Are all branch conditions correct?*	[BEA94]	
L.19.C	Are loop index variables used only within the loop?	[NAS94]	
L.20.C	Are all loops free from loop index modification?	[NAS94]	
L.21.C	*Is all loop nesting in the correct order?*	[NAS94]]	
L.22.U	Do all lops have single exit and entry points?	[NAS94]	5.6.4.e
L.23.U	For all nested loops, are loops and loop exists labeled?	[SPC91]	5.1.1.a, 5.1.3.a
L.24.C	Is the ternary conditional operator "?:" used correctly?	[POT94] p. 174	
L.25.C	Are the increment and decrement operators properly used in postfix and prefix order?	[POT94] p. 177	
L.26.U	Do braces surround the body of a "for" and "while" loop even though it only has one statement (i.e., exhibiting good programming practices)?	[POT94] p. 204, 228	
L.27.U	Are the expected executions anticipated with "while", "do while", and "if while", even though the code will compile?	[POT94] p. 204, 209, 212	
L.28.C	Are "exit (status)", "break in case", and "break and continue" used to correctly exit the program or exit the loop?	[POT94] p. 212, 213, 252, 260	
L.29.C	Are counters initialized to zero and the increment operator (i.e., "++") used appropriately?	[POT94] p. 218	
L.30.C	When "for" loops are used, is the intent for the condition to be tested at the top of the loop (i.e., is the condition ever "True" so that the loop executes)?	[POT94] p. 229	
L.31.C	Is redundancy eliminated in "for" loops for better efficiency?	[POT94] p. 235	
L.32.C	Do all "switch" statements contain a default branch to handle unexpected cases?	[BAC92] p. 12 [DST96]	
L.33.C	Does logic handle bad input as well as good input?	[BEA94]	

• Other •

Identifier	*Question*	Source	Guideline
O.01.U	*Is the descriptive prologue complete and correct?*	[BEA94]	
O.02.C	*Are all printed or displayed messages free from grammatical or spelling errors?*	EBE94]	
O.03.U	*Does the code follow basic structured programming techniques?*	[NAS94]	
O.04.U	*Are all assumptions documented?*	[NAS94]	
O.05.C	*Is the code written only in Ada?* Is the code written only in C or C++?	[SQF95] AP.3.4.e	7.6.3.a
O.06.U	Is each variable declared on a single line to improve readability and maintainability?	[POT94] p. 55	
O.07.U	Does code contain mapping to parent documents, or functional specifications?	[DST96]	

A sample checklist blank data collection form is included at the end of Appendix B. This sample checklist is identical to the one used in the second Field Trial.

3.3 Automated Ada Style Guideline Checks

The objective of this task was to develop an automated static analysis tool that would identify style guideline violations in Ada code. The tool would then be used in the static analysis step of the first certification field trial. Under the previous RC-SQF effort, GRCI's AdaQuest tool was enhanced to perform automated data collection for a set of SQF questions. Some of these questions were style guideline checks. Under this effort the AdaQuest tool was enhanced to provide a more comprehensive set of style guideline checks, and the user interface and reports were modified to suit the certification process.

The result of this task was AdaQuest version 2.2, which was successfully used in the first certification field trial. In the certification field trial, style guideline violations were considered "minor" defects. In an application where portability is a concern, however, the portability-related violations would be considered "major" defects.

The assessment of compliance to coding style standards or guidelines is part of the code inspection method embodied in SQF 1.5. In the SQF data collection form, it is represented as a single question to be answered for each inspected unit:

```
    Does this unit comply with coding standards?
```

Answering this single question implies a great deal of inspection effort. It is most cost effective to automate as much of this type of code inspection as possible. A single yes or no answer, however, is not sufficient to enable corrective action, that is, correction of the deficiencies. In order to fix the problem, a tool must report the line(s) of code and the nature of the problem.

Implemented Style Checks

For Ada code, the style guidelines documented in the SPC Ada 95 Quality and Style [SPC95] have been adopted by the Ada Joint Program Office and are widely accepted as a standard. The new AdaQuest Auditor supports the style guideline checks and coding practices listed in Table 3-4 [GRC95]. Where directly traceable to [SPC95], an SPC reference number is given. "SPC 5.6.4.d" for example, refers to the fourth bullet under section 5.6.4.

It is important to note that [SPC95] is organized into chapters of related guidelines. AdaQuest 2.2 has implemented checks from the four chapters listed below. A total of 45 style checks were implemented; Figure 3-8 shows the breakdown by chapter. Most of the implemented style checks fall into the programming practices category. The "other" category includes checks derived from the SQF but not traceable to [SPC95] and a check of cyclomatic complexity versus a threshold value of 10.

Chapter 3	Readability
Chapter 4	Program Structure
Chapter 5	Programming Practices
Chapter 7	Portability

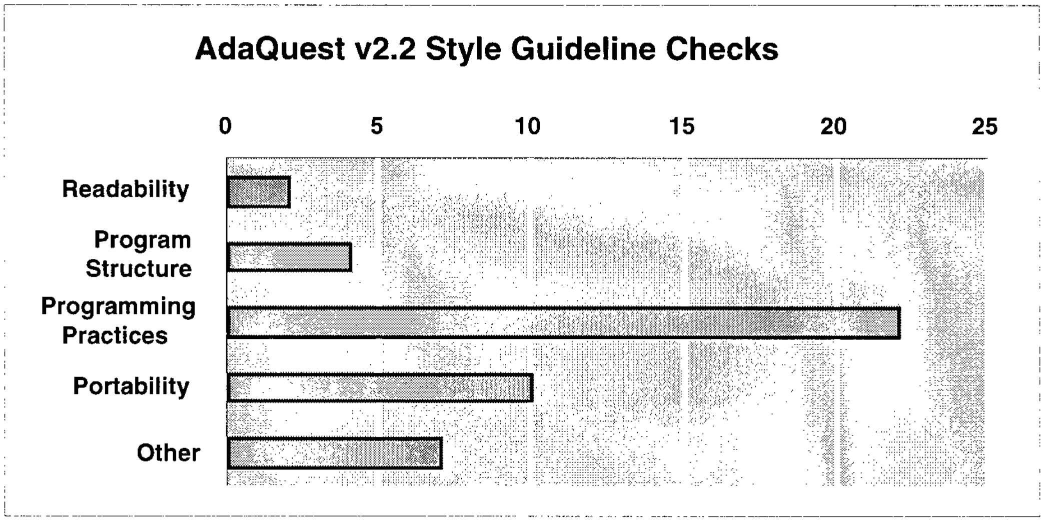


Figure 3-8. Summary of Implemented Style Guideline Checks in AdaQuest 2.2.

AdaQuest 2.2 allows the user to specify a set of checks to disable in order to customize the inspection for a particular project or organization's standards. For example, if portability is not a concern for a project, the chapter 7 checks could be disabled.

Table 3-4. AdaQuest Version 2.2 Auditor Checks

Violation Identifier	Reference	Description
Anonymous_Types	SPC 5.3.2.a	Avoid anonymous types
Case_Without_Others	SPC 5.6.3.1	Do not use a "when others" choice in a CASE statement - explicitly enumerate each possible choice
Case_Without_Ranges	SPC 5.6.3.b	Do not use ranges of enumeration literals in CASE statements - explicitly enumerate each choice instead
Code_Statement	SPC 7.6.3.a	Avoid machine code inserts
Comp_Unit_Not_In_Separate_File	SPC 4.1.1.a,e	Place each compilation unit in a separate file
Conditional_Exit	SPC 5.6.5.b	Use "exit when..." rather than "if...then exit"
Constrained_Discriminant_Records		Discriminant record with defaults - subtype can be changed after elaboration
Constraint_Error_Disabled	SPC 5.9.5.a	Remove pragma Suppress - be prepared for Constraint_Error exceptions after removal
Control_Parameter_Reference	SQF MO.1.6	Interface parameter is used as a control variable
Cyclomatic_Complexity		Cyclomatic complexity > 10
Exits_In_Simple_Loops	SPC 5.6.4.d	Avoid exit statements in WHILE and FOR loops
Explicit_In_Parameters	SPC 5.2.4.a	Show the mode indication of IN parameters
Explicit_Subtypes	SPC 3.4.1.d	Avoid "anonymous subtypes" (type and constraint) - declare and use a subtype, if possible
For_Loops_With_Type		Avoid ranges of the form X..Y in a FOR loop - use a type/subtype name or 'RANGE instead
Formal_Part_Defaults_Not_Last	SPC 5.2.3.b	Place default parameters at the end of the formal parameter list
Global_Variable_Declared	SPC 4.1.8.a	Avoid declaring variables in package specifications
Global_Variable_Used	SQF MO.1.7	Do not read from global variables
Global_Variable_Updated	SQF MO.1.8	Do not write to global variables
Goto_Statement	SPC 5.6.7.a	Do not use GOTO statements
Handler_Without_Others	SPC 5.8.2.a	Use "when others" handlers with caution - better to explicitly name each exception that can be encountered

Implementation_Defined_Attribute	SPC 7.6.5.a	Avoid attributes added by the compiler implementer
Implementation_Defined_Pragma	SPC 7.6.5.a	Avoid pragmas added by the compiler implementer
Library_Body_Has_No_Matching_Spec	SPC 4.1.1.b	Create an explicit specification, in a separate file, for each library unit subprogram
Named_Association_In_Aggregates	SPC 5.6.10.c	Use named association for aggregates unless there is a conventional ordering of the arguments
Named_Associaton_In_Instantiations	SPC 5.2.2.b	Use named association when instantiating generics
Named_Blocks	SPC 5.1.2.a	Associate names with blocks when they are nested
Named_Ends	SPC 5.1.4.a	Repeat the simple name at the end of a block structure
Named_Exits	SPC 5.1.3.a	Use loop names in EXIT statements that exit nested loops
Named_Loops	SPC 5.1.1.a	Associate names with loops when they are nested
Non_Scalar_Types_Private	SPC 5.3.3	Non-scalar full type in package spec - can it be PRIVATE or LIMITED PRIVATE?
Numeric_Error_Disabled	SPC 5.9.5.a	Remove pragma Suppress - be prepared for Numeric_Error or Constraint_Error exceptions after removal
Pragma_Interface	SPC 7.6.4.a	Avoid interfacing Ada to other languages
Predefined_Identifier_Redefined	SPC 3.2.2.e, 3.4.1.c	Do not redefine names from package Standard
Program_Error_Disabled	SPC 5.9.5.a	Remove pragma Suppress - be prepared for Program_Error exceptions after removal
Real_Operand_Comparison	SPC 5.5.6.a, 7.2.7.a	Avoid ″=″, ″/=″, ″<″, or ″>″ when comparing real values - use ″<=″ or ″>=″ instead
Representation_Clause	SPC 7.6.1.a	Avoid the use of representation clauses
Short_Circuit_Operators	SPC 5.5.5.a	Use the short circuit forms of logical operators (AND THEN instead of AND, OR ELSE instead of OR)
Single_Identifier_Lists		Declaration with multiple identifiers - split into separate declarations with one identifier each

Storage_Error_Disabled	SPC 5.9.5.a	Remove pragma Suppress - be prepared for Storage_Error exceptions after removal
Unchecked_Conversion	SPC 5.9.1.a, 7.6.8.a	Avoid using Unchecked_Conversion
Unchecked_Deallocation	SPC 5.9.2.a, 7.6.6.a	Restrict the use of Unchecked_Deallocation
Use_Clause	SPC 5.7.1.b	Minimize using the USE clause
Use_Of_Predefined_Numeric_Type	SPC 7.2.1.a	Do not use the predefined numeric types from package Standard
Use_Of_System_Constant	SPC 7.6.2.a	Avoid using package System constants
User_Exceptions_Raised	SPC 4.3.1.j	Do not explicitly raise predefined exceptions

AdaQuest User Interface and Reports

AdaQuest 2.2 is integrated into the Rational Apex Ada development environment via the Ada Semantic Interface Specification (ASIS). ASIS is a vendor-independent interface to the semantic contents of Ada libraries and provides the comprehensive information needed for AdaQuest static analysis. AdaQuest analyses can be invoked from the Apex menus, and the Apex graphical user interface provides a display and control integration mechanism.

Code to be audited by AdaQuest must first be imported and compiled into the Apex environment. In an Apex window, units to be analyzed are selected and then AdaQuest is launched from the Apex Tools menu. When the AdaQuest analysis is complete, the results appear in a new window which list violations of style guidelines for each unit analyzed.

The Apex integration provides a very useful feature in AdaQuest: a direct link to the specific line of code where the violation was found. To browse the code, the user simply selects the violation message and uses the Visit button to bring up an Apex editor window with the faulty line of code highlighted. This provides a corrective action mechanism, linking the detection of a style guideline violation with the means to fix the problem—editing the code. This linkage helps to prevent defects by allowing the code developer to easily apply a "self test" to his code while it is still within his control, and *before* it is checked in for integration and testing.

3.4 Guidance for Building Correctness into Code Assets

The objective of this task was to extract guidance for building the quality factor of Correctness into reusable code assets. The guidance was generated by reverse-engineering the SQF using the approach discussed in section 2.5. The results are documented in this section.

The quality factor Correctness deals with "the extent to which software design and implementation conforms to specifications and standards". The IEEE dictionary [IEE91] offers three definitions shown below. The SQF definition of *Correctness* most closely matches the second IEEE definition.

Factor	Definition
Correctness	(1) The degree to which a system or component is free from faults in its specification , design, and implementation. (2) The degree to which software, documentation, or other items meet specified requirements. (3) The degree to which software, documentation, or other items meet user needs and expectations, whether specified or not.

In the SQF hierarchical structure, Correctness decomposes into the following three criteria:

Criterion	Definition
Completeness	Those characteristics of software which provide full implementation of the functions required.
Consistency	Those characteristics of software which provide for uniform design and implementation techniques and notation.
Traceability	Those characteristics of software which provide a thread of origin from the implementation to the requirements with respect to the specified development envelope and operational environment.

The SQF 1.5 view of Correctness emphasizes product characteristics as opposed to process. For example, SQF 1.5 does not ask about satisfactory completion of testing, or deal with test stopping criteria (such as coverage of requirements or of structure) as part of Correctness. One exception to the product focus are the questions that deal with closure of problem reports.

The SQF 1.5 view of Correctness is also limited to functional requirements as opposed to quality requirements. Using the SQF approach, quality requirements are treated as separate quality factors rather than as an aspect of Correctness. The SQF 1.5 view deals with internal characteristics of the software, however, rather than behavior.

A developer's approach to ensuring Correctness would not be complete without consideration of testing. Therefore the SQF approach is really an adjunct to what would be considered typical quality assurance activities, and is not a replacement for them. The guidance in this section has supplemented the SQF guidance to provide a more complete approach, including indicators that give the project manager or development engineer insight into achievement of correctness.

Usage Scenarios for Correctness

There are two main usage scenarios for the factor Correctness:

1) Development of new software

2) Evaluation of the correctness of existing software.

For the development of new software, Correctness is a given in almost every situation; it is difficult to imagine a case where Correctness is not a concern. There may be cases, such as with throwaway prototypes, where ensuring Correctness by inspecting the software would require more effort than is warranted.

The second scenario seeks to determine the Correctness of an existing software system, or part of a system, but not during the development process. The reason for this evaluation may be one of the following situations:

- Verification and validation by an independent organization (IV&V)
- Evaluation of test readiness
- Selection of reusable or COTS components.

For the last situation listed above, an important aspect of the evaluation would be adherence to standards.

Correctness Guidelines

We have extracted guidelines from the data collection forms of SQF version 1.5 and supplemented these with additional guidelines we felt were needed to round out the SQF approach to Correctness. Table 3-5 summarizes the resulting Correctness guidelines.

Of the 14 guidelines, 12 were extracted from SQF 1.5, and the specific question identifier is noted in the table.

Each guideline has one or more designated products and references. A designated product provides tailoring assistance and helps to link the guideline to the software development process. References provide validation of the guideline as well as a source for more detailed information.

Table 3-5. Correctness Guidelines

Id	Guideline	Product(s)	Reference(s)
C_001	All capabilities and functional entities (units) should have inputs, processing and outputs defined (i.e., documented with source, meaning, and format).	Software Requirements Software Design Description Source Code	[SQF95] CP.1.1,8
C_002	All data items should be defined (i.e., documented with source, meaning and format).	Software Requirements Software Design Description Source Code	[SQF95] CP.1.3
C_003	All capabilities should be allocated to the next level of decomposition.	Software Requirements Software Architecture Software Design Description Source Code	[SQF95] CP.1.7
C_004	All capabilities or functional entities (units) should have the processing flow (algorithms) and decision points (conditions & alternative paths) described.	Software Requirements Software Architecture Software Design Description Source Code	[SQF95] CP.1.10
C_005	All problem reports should be resolved.	Correction Problem Reported Info.	[SQF95] CP.1.12
C_006	There should be no superfluous parameters in the argument list; all parameters should be used in the unit.	Software Design Description Source Code	[SQF95] CP.1.11
C_007	The following items should be standardized: design representations, calling sequence protocol, external I/O protocol, error propagation & handling, references to data, data representation within the design, data naming conventions, and definition & use of global variables.	Software Architecture Software Design Description Source Code	[SQF95] CS.1.1,4,7, CS.1.10, CS.2.2, CS.2.4, CS.2.7
C_008	All references to capabilities, units, and data items should use a single unique name.	Software Requirements Software Architecture Software Design Description Source Code	[SQF95] CS.1.12
C_009	If multiple copies of the same information are required (e.g. copies at different nodes), there should be procedures to establish the consistency & concurrency of them.	Software Requirements Software Architecture Software Design Description Source Code	[SQF95] CS.2.11
C_010	There should be no superfluous requirements; all allocated requirements should be traceable to the parent entity (system, subsystem, CSCI, etc.).	Software Requirements Software Architecture Software Design Description Source Code	[SQF95] TC.1.1
C_011	Each unit's description should identify all requirements that it helps to satisfy.	Software Design Description Source Code	[SQF95] TC.1.2
C_012	All required capabilities should be implemented.	Software Design Description Source Code	
C_013	All requirements should be tested.	Test Planned Info. Test Summary Reported Info.	
C_014	Testing exit criteria should be established.	Test Planned Info.	

Evaluation of Correctness

The guidelines were then broken down into inspection items and indicators to support evaluation of the factor Correctness, shown in Table 3-6. Like much of the existing SQF (version 1.5) data collection forms, most guidelines are converted into questions that are answered at a product inspection.

Like other problems found at inspections, SQF-derived problems are documented and handled as part of the normal defect reporting and tracking mechanism. Correction of these defects is referred to as rework.

Table 3-6. Correctness Inspection Items and Measures

Id	Guideline	Inspection Items	Measures
C_001	All capabilities and functional entities (units) should have inputs, processing and outputs defined (i.e., documented with source, meaning, and format).	Do all capabilities have inputs, processing and outputs defined? Does this unit have inputs, processing and outputs defined?	# units successfully passing inspection total # units
C_002	All data items should be defined (i.e., documented with source, meaning and format).	Have all data items been defined?	
C_003	All capabilities should be allocated to the next level of decomposition.	Have all capabilities been allocated to the next level of decomposition?	
C_004	All capabilities or functional entities (units) should have the processing flow (algorithms) and decision points (conditions & alternative paths) described.	Do all capabilities have processing flow and decision points described? Does this unit have processing flow and decision points described?	
C_005	All problem reports should be resolved.		# open problem reports # closed problem reports
C_006	There should be no superfluous parameters in the argument list; all parameters should be used in the unit.	Are all of this unit's parameters used in the unit?	
C_007	The following things should be standardized: design representations, calling sequence protocol, external I/O protocol, error propagation & handling, references to data, data representation within the design, data naming conventions, and definition & use of global variables.	Is there a standard for design representation? for calling sequence protocol? for external I/O protocol? for error propagation & handling? Have conventions been established for data references? for data representation in the design? for data naming ? definition & use of global variables? Does this unit adhere to the established standards for ...(see above list)?	
C_008	All references to capabilities, units, and data items should use a single unique name.	Do all references to capabilities, units, and data items use a single unique name?	
C_009	If multiple copies of the same information are required (e.g. copies at different nodes), there should be procedures to establish the consistency & concurrency of them.	Are there procedures for establishing the consistency & concurrency of multiple copies of the same information?	

C_010	There should be no superfluous requirements; all allocated requirements should be traceable to the parent entity (system, subsystem, CSCI, etc.).	Are all allocated requirements traceable to the parent entity?	for each entity, # requirements untraceable to parent entity total # requirements
C_011	Each unit's description should identify all requirements that it helps to satisfy.	Does this unit's description identify all requirements that it helps to satisfy?	
C_012	All required capabilities should be implemented.	Have all capabilities been incorporated into the design? Has the design been completely implemented?	# capabilities untraceable to design # capabilities untraceable to code
C_013	All requirements should be tested.	Are all requirements traceable to test cases? Have all tests been successfully completed?	# requirements untraceable to test cases # tests successfully passed total # tests
C_014	Testing exit criteria should be established.	Have exit criteria for testing been established?	

Correctness Indicators

The most common indicator of Correctness is defects, or the lack thereof. Defect density is compared to established norms to determine if the software development is proceeding as expected and to evaluate the effectiveness of defect removal activities. An example of a defect density trend indicator is shown in Figure 3-9. Defect density is measured periodically throughout the development life cycle and compared to the expected range of values determined from historical data.

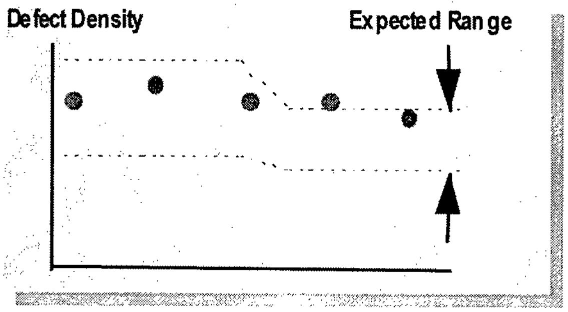


Figure 3-9. Defect Density Indicator

Closure of problem reports of specified severity levels is typically used as an acceptance criterion, or an exit criterion before proceeding on to the next phase. Figure 3-10 shows an example of a problem report closure indicator. The cumulative total number of problem reports with the status of "open" and "closed" are plotted over time. The gap between the "open" and "closed" curves represents the backlog of rework that must be addressed to achieve Correctness. When the gap closes, all open reports have been closed, and this is a measure of Completeness.

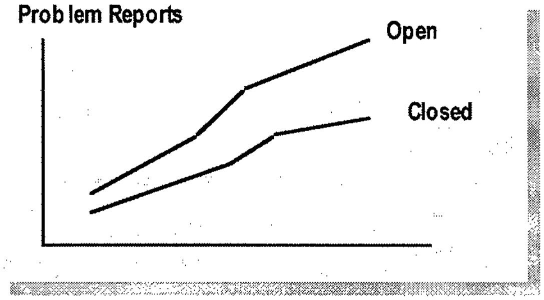


Figure 3-10. Problem Report Closure Indicator

Completeness is typically viewed by work unit progress type indicators. Units having successfully passed milestones (such as detailed design, implementation, inspection, unit testing, and integration) would be counted for a development progress indicator. Figures 3-11 and 3-12 are examples of work progress indicators. In addition to indicating progress toward completion, plotting units passing inspection also indicates compliance to the quality assurance (QA) process. Experience has shown that projects that abandon inspection in order to make up schedule end up taking more time during testing and repair of defects found during testing.

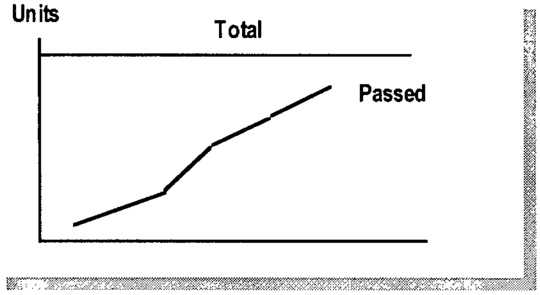


Figure 3-11. Work Progress Indicator for Units Passing Inspection

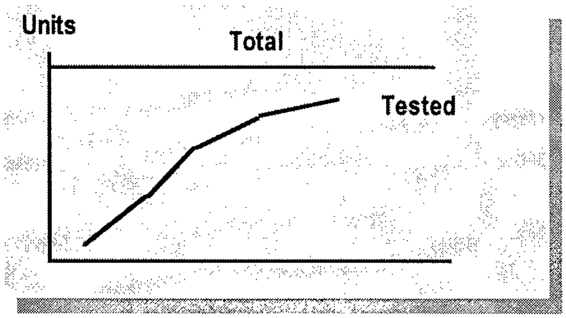


Figure 3-12. Work Progress Indicator for Units Passing Testing

Traceability indicators focus on "holes" in a traceability matrix: what proportion of software requirements have not been completely implemented? which units are not traceable to a software requirement?, etc.. Figure 3-13 shows an example of how this data is typically presented. Each column on the chart represents the number of *untraced* requirements from one document to another, such as from the system requirements to the software requirements specification, or from design to implementation. Each group of columns represents the date at which the status is reported, which could be monthly, quarterly, or at milestones. As progress is made, the height of the columns should decrease over time. By focusing on untraced requirements rather than traced, this indicator emphasizes problem areas and work to be done in order to achieve Correctness.

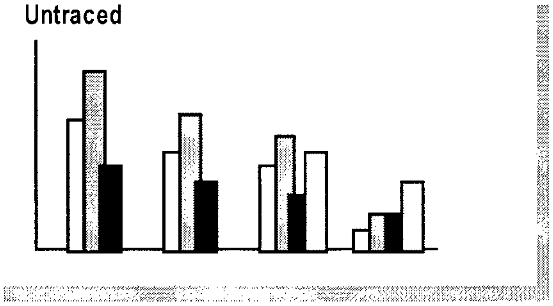


Figure 3-13. Traceability Indicator

Another useful indicator of traceability data, shown in Figure 3-14, is a plot of the number of requirements that have been (successfully) tested versus time. This not only shows testing progress, but also indicates progress toward Completeness of the implementation of all requirements.

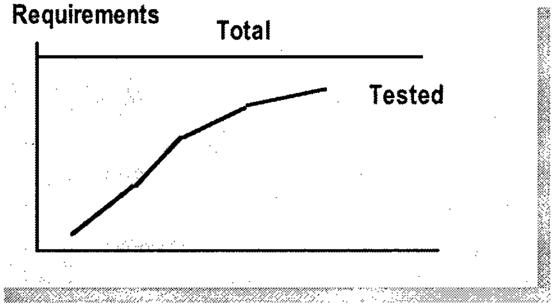


Figure 3-14. Test Completeness Indicator

3.5 Predictive Model

The MITRE effort, contract F19628-94-C-0001, included a task entitled Predictive Models for Categorizing Error-Prone Software Modules. This task developed a series of predictive models to estimate the number of defects in a module based on characteristics of the module and of the environment in which it was developed. Information in this section is excerpted from the task Final Technical Report [THO96].

For the purposes of these models, a module is defined as a library unit aggregation (LUA), which is defined as the collection of an Ada library unit and all of its associated secondary units. The models were calibrated based on six Ada programs, ranging in size from 35 to 75 KSLOC developed in the Flight Dynamics Division at NASA/GSFC. See [THO96] for more details.

The initial model predicted the total number of defects. This model used the explanatory variables listed in Table 3-7.

Table 3-7. Explanatory Variables for Total Defects Prediction

Type	Explanatory Variable	Definition
Module Characteristics	Size	Number of program unit declarations
	Context Coupling	The ratio of imported program unit declarations to the number of exported program unit declarations
	Control Coupling	Number of calls per subprogram
	Workload Complexity	Average parameters per visible subprogram
Environment Characteristics	Development Volatility	Non-defect changes/KSLOC
	Reuse	Percent custom code

The model was refined into a series of models, one for each defect type (i.e., computational, data, interface, and logic). The defect category "other" was not modeled. Explanatory variables that measure types of complexity were added; these are summarized in Table 3-8. All added variables are characteristics of the module.

Table 3-8. Added Explanatory Variables for Prediction of Defects by Type

Type	Explanatory Variable	Definition
Module Characteristics	Cyclomatic Complexity	Average cyclomatic complexity of all subprograms in an LUA. Cyclomatic complexity is the number of independent paths in a control flow diagram.
	Arithmetic Complexity	Number of arithmetic operations per module
	Data Declaration Complexity	Relative count of constant and number declarations
	Relational Complexity	Ratio of the number of relational operations per cyclomatic complexity
	Exception Use	Number of programmer-defined exceptions

Table 3-9 shows which of the explanatory variables were found to be applicable to prediction of the different defect types.

Table 3-9. Relevance of Explanatory Variables to Defect Types

	Defect Type			
Variable	**Logic**	**Data/Value/Init**	**Interface**	**Computational**
Size	✓	✓	✓	✓
Context Coupling			✓	
Control Coupling		✓	✓	✓
Cyclomatic Complexity	✓			
Workload Complexity		✓	✓	
Arithmetic Complexity				✓
Data Declaration Complexity		✓		
Relational Complexity	✓			
Exception Use	✓		✓	
Development Volatility	✓	✓	✓	✓
Reuse	✓	✓	✓	✓

These defect models were found to be effective at predicting the number of defects by type. In fact, a model consisting of the sum of the predicted defects by type was found to be slightly better at predicting the total number of defects than the original model (based solely on the variables in Table 3-7). Thus our hypothesis about the value of complexity measures in predicting different types of defects, as introduced in section 3.1.4, was confirmed.

4.0 Expansion of the Rome Laboratory Certification Framework

In order to met our objective to further develop, apply and validate the Rome Laboratory Certification Framework (CF) intially developed under the CRC contract, we chose to expand the CF by applying it to multiple domains and to a commercial reuse organization as an additional pilot site. Additional CF visibility was achieved by the CRC Web pages. All these activities are described in the subsections that follow.

4.1 Field Trial and Pilot Studies

As part of the expansion of the CF, an additional field trial of a C++ asset and pilot studies at the Air Force Reuse Center were conducted as described below.

4.1.1 Field Trial with a C++ Code Asset

In an effort to expand the RL Certification Framework, a second Field Trial was conducted using the same generic certification process. Field Trial #2 was conducted with the following purposes:

- Ensure the certification process is repeatable
- Assure the certification process is understandable by other certifiers
- Collect data on the effort required to perform the certification process
- Collect data on the effectiveness of the techniques defined in the Certification Framework

The default certification process used for both Field Trail #1 and Field Trial #2 is shown in Figure 4-1.

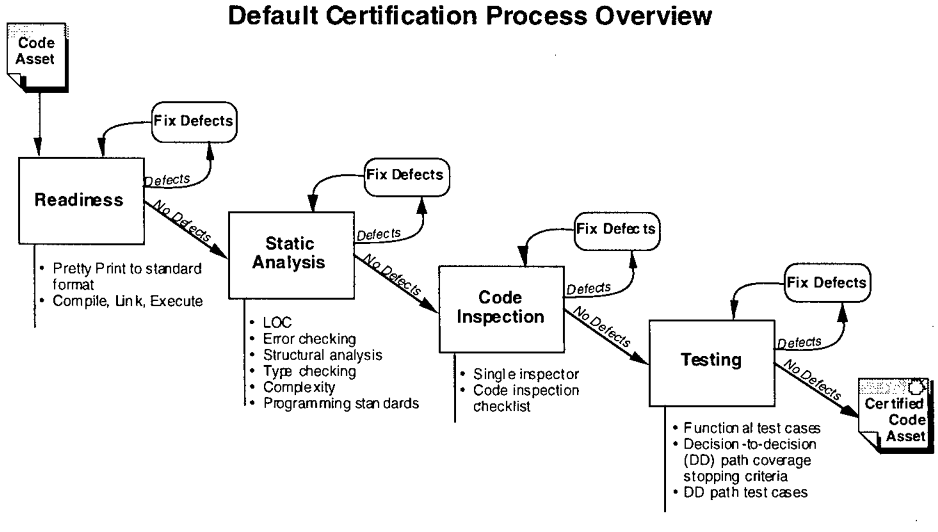


Figure 4-1. Default Certification Process Used in the Field Trial

Two different roles were required to perform the field trial activities. The first role was that of Defect Seeded and Data Analyzer. The second role was that of Certifier. The defects were seeded without the knowledge of the Certifier.

The following major activities were performed to conduct Field Trial #2:

1. Select hardware and software tools, install, configure, and integrate.
2. Revise certification instruments.
3. Select source code component.
4. Establish component baseline.
5. Seed defects.
6. Perform the Certification Field Trial.
7. Collect Data.
8. Analyze data and results.
9. Compare Field Trials #2 results with Field Trial #1.
10. Document results of Field Trial #2 in ATD FTR Volume 1 and ATD FTR Volume 2.

The timing and duration of effort for these activities are illustrated in Figure 4-2. As seen in this illustration, the field trial itself is but a small part of the overall activities that are required.

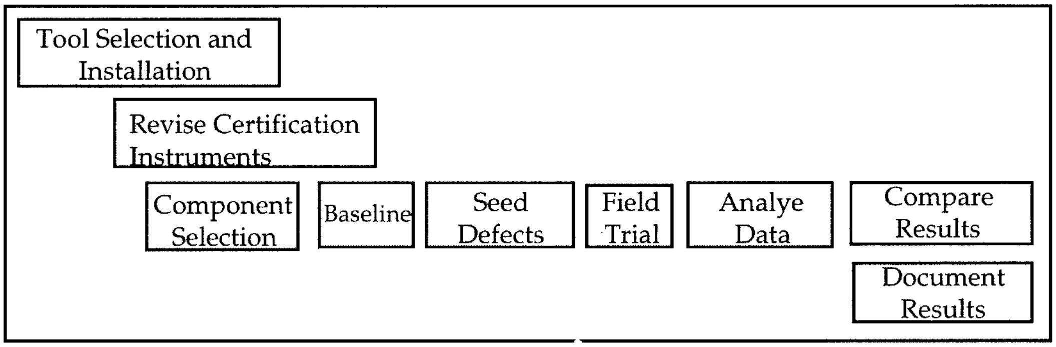


Figure 4-2. Timing and Effort of Activities

The actual Field Trial #2 itself was scoped to a two person week effort to maintain consistency with Field Trial #1. Field Trial #2 used a C++ code asset of about 1,000 logical lines of code. Certifying this size of asset was estimated at a 2 person-week effort. This time period was indicated by staff at reuse libraries as that which was available to perform such an activity. Field Trial #2 used the PC environment, and the asset was certified for the same factors as Field Trial #1 (i.e., Correctness, Completeness, Understandability).

A PC platform was selected for Field Trial #2. We used a Dell Latitude LM P133ST Portable with Docking Station, 133 MHz Pentium Processor, 40MB RAM, 1.3 GB Hard Drive, 14" SVGA Active Matrix Color Display, CD-ROM and Floppy Drive. The PC platform was running MS-DOS and Windows 95.

We used the CRC Certification Tool Requirements, selection process and C/C++ Tool Survey found in CRC's Volume 6. We also performed an additional cursory look outside the survey for any new tools or upgrades that arrived on the COTS market since the tool survey was conducted.

We selected the Borland C++ 5.00 Interactive Development Environment (IDE) as our compiler for Step 1. Readiness. During the field trial, it was upgraded to version 5.01 to achieve compatibility with other certification tools. PC-Lint 7.0 was used for Step 2. Static Analysis. C-Vision 4.0 was used for Steps 2 and 3, Static Analysis and Code Inspection, respectively. The McCabe Visual Toolset 5.2 was selected for Step 2 and Step 4, Static Analysis and Testing, respectively. The McCabe Visual Toolset was upgraded to version 5.2.2 to achieve compatibility with the Borland compiler during instrumentation of the code.

The default certification process was analyzed and no process modifications were needed to accommodate the characteristics of the C++ component. The same data collection plan, procedures, and forms were used to maintain consistency across field trials. The Ada Code Inspection Checklist was modified to accommodate a C++ component; the results of this activity appears in section 3.2.2 of this document.

To selection the code component, the following criteria were used:

- Is a utility-type program
- Is ~1000 Lines of Code (excludes standard "includes", other "includes" counted once)
- Can be certified in two weeks of effort
- Is functional as stand-alone, is a complete "system" as a single executable image
- Has computational characteristics
- Generation of test drivers requires limited effort
- Execution of component produces recognizable results for ease of testing
- Provides adequate documentation
- Compiles pre- and post- defect seeded

A example component package with the Borland IDE was selected because is satisfied our criteria. The selected component was a mailing labels program. It updates and displays the contents of a mailing list. New subscriptions can be input, and the user is notified if subscriptions have expired. The program has data input at the unit level, and it has file input and output. It does not have a graphical user interface, and runs from the MS-DOS prompt. The following other code components were considered during the selection process:

- Tools.h++ with Test Suite - COTS Standard C++ Library by Rogue Wave
- Two example components packaged with McCabe Visual Toolset
- Eighteen other code examples packaged with the Borland compiler
- Public domain components (e.g., C/C++ utility components on the World Wide Web)
- Government/Commercial Repositories
- SPS internally-developed code components

Each of the above were eliminated as candidates as defined by our selection criteria.

The component was baselined and seeded defects were added. Defect reports were stored on a Macintosh computer in a FileMaker Pro file. This method provided the ability to easily document, number, sort, find, and count the reports. A thorough analysis of the results was facilitated with these tools. Similar analyses were performed for the second field trial to maintain continuity with the first. This also permitted a comparison of data across both field trials. Details about the seeded and natural defects and the entire Field Trial #2 can be found in the ATD FTR, Volume 2- Additional Certification Field Trial. A summary of the results of the Certification Field Trial for a C++ Asset follows.

Field Trial Overview

The certification field trial described in this report was performed by SPS personnel Sharon Rohde, Pat Aymond and Karen Dyson.

Personnel. Ms. Rohde was selected to perform the field trial because of her experience with the C++ language, and also because she was not involved in the derivation of the default certification process or in writing the field trial procedures. Ms. Rohde installed the certification tools and performed all of the certification steps.

Ms. Dyson was a contributor to the derivation of the default certification process and co-author of the field trial procedures. Pat Aymond selected the asset to certify and seeded additional defects into the asset, consulting with Karen Dyson. Ms. Dyson served as consultant for the analysis of the results and lessons learned.

Objectives

The objectives of the field trial were as follows:

- Perform all of the steps in the default certification process
- Use all of the tools in the certification tool set
- Assess the accuracy and understandability of the procedures guidance
- Collect effort and technique effectiveness data
- Select a single asset to certify sized for a 2 staff-week certification effort

While technique effectiveness data was collected, the field trial was not intended to be an experiment to determine the effectiveness of the techniques that comprise the default certification process. The design and implementation of an experiment of that type is quite involved and is significantly beyond the scope of the CRC/ATD contract. The effort and technique effectiveness data was collected in order to compare the actual results with comparable values culled from other research studies.

Accomplishments

All of the above objectives were satisfied by the field trial.

Asset Certified

The resources allocated to the field trial task allowed for certification of a single asset. The asset to certify was selected based on its similarity with the asset certified in the Ada field trial. Since the default certification process was derived for Ada code assets, it was modified for a C++ code asset. The Reuse Code Inspection Checklist was modified for a C++ code asset.

Size. It was estimated that an asset of about 1000 logical lines of code would be large enough to not be trivial and yet small enough to be certified in a 2 staff-week effort. The effort constraint was developed based on extensive interviews of reuse library personnel performed early in the CRC contract [see the CRC Volume 4 - Operational Concept Document], which indicated that 2 staff-weeks were about the right amount to devote to certifying a single asset.

Defect history. In order to assess the effectiveness of the certification process at finding defects, it was necessary to have an asset with defects known in advance. To achieve this requirement of a defect history, we seeded defects into the selected component to the similar extent as the Ada component in the previous initial field trial.

Selected Asset

The selected asset was a labels program packaged with the Borland compiler as example code. This single executable program automatically generates mailing labels from a master list. It reads a subscription list, inserts new subscriptions into a master list, and prints the contents of the master list in a standard label format. It had no recorded defect history.

Size of Asset

Lines of code (physical)	3,356 lines (est.)
Number of class libraries	3
Number of supporting "C"files	2

The size of the asset was determined by counting lines which included compiler directives and the following header files with seeded defects:

- <classlib\listimp.h>
- <classlib\objstrm.h>
- <classlib\date.h>

An informal desk check type code review turned up no major or minor defects. Therefore we decided to seed 14 additional major defects in order to have a significant number of major defects known in advance of the field trial. The seeded defects are summarized in the table below. All seeded defects are documented in FTR Volume 2 - Additional Certification Field Trial, Appendix B and have an identifier starting with "PA_". These known defects were not shown to Ms. Rohde prior to or during the field trial.

Summary of Seeded Defects

Identifier	Unit	Lines	Description	Type
PA_001	labels.cpp	386-387	Changed write to read output file in subscription list destructor. Does not write subscriptions to master file.	Interface
PA_002	date.h	31	Changed value of constant Julian date of 1/1/1901 from "2415386L" to "1415386L"	Data
PA_003	date.h	106	Changed operator "-=" to "=" and data type from integer to constant.	Interface
PA_004	date.h	253,256	Changed operator "-=" to "=" in inline operator definition.	Interface
PA_005	date.h	272	Changed inline function that checks for valid months so that months January and December are not valid.	Logic
PA_006	listimp.h	82,83,91	Instead of zeroing out the list element counter, it was set to 1.	Logic
PA_007	listimp.h	719	In ForEach function, incorrect while condition does not iterate through list properly.	Logic
PA_008	listimp.h	889	Changed notation from class name to arithmetic operator.	Logic
PA_009	objstrm.h	299	Address of object is not stored in database.	Logic
PA_010	objstrm.h	626	Changed inline function clear, changed "hardfail" to "basefield".	Data
PA_011	objstrm.h	1020	Improper terminator in switch statement; changed "break" to "switch".	Logic
PA_012	labels.cpp	414	Wrong while loop condition, changed "iter !=0 to "iter == 7". Will not correctly write subscription list to output file.	Logic
PA_013	labels.cpp	429	Incorrect initialization of for loop iterator; changed "i = 0" to "i = 11". Will not read in subscriptions from master file unless count > 11.	Logic
PA_014	labels.cpp	605	Changed type declaration of main routine from "int" to "unsigned int".	Interface

The seeded defects were not created in an attempt to duplicate a particular defect profile (i.e., distribution of defect types). There are more logic defects than other types simply because these are the easiest type to invent. It turned out to be rather more difficult than we anticipated to create defects that were not caught by the compiler, nor caused immediate catastrophic failure on execution.

In the results section below, we look at all of the known defects in the certified asset after having completed the certification process.

Certification Results

This subsection presents the results of the second certification field trial performed by SPS. Analysis of the data collected during the field trial and of the defects found in the asset are included in these results. Lessons learned are discussed in the next subsection.

Data collection forms described in below were completed during the field trial. All certification defect reports are in Volume 2 - Additional Certification Field Trial, Appendix B, and the other completed forms are contained in this subsection under the appropriate topic.

Staff Experience

As mentioned the overview above, three SPS personnel were involved in the field trial. Their completed Certifier Profile Worksheets follow.

CERTIFIER PROFILE WORKSHEET

CERTIFIER NAME OR ID NUMBER	Sharon Rohde
Number of years of programming experience	5 yrs
Number of years of programming experience in asset's language	.5 yrs in C++
Education (list degrees)	MS Computer Science
Experience with Certification Tools (hours with each tool before starting certification process)	
Borland C++ IDE	10 hr
PC-Lint	3 hrs
McCabe Visual Toolset	32 hrs
C-Vision	8 hrs

CERTIFIER NAME OR ID NUMBER	Pat Aymond
Number of years of programming experience	10 yrs
Number of years of programming experience in asset's language	5 yrs
Education (list degrees)	MS, Education
Experience with Certification Tools (hours with each tool before starting certification process)	
Borland C++ IDE	2 yrs
PC-Lint	0
McCabe Visual Toolset	0
C-Vision	0

CERTIFIER NAME OR ID NUMBER	Karen Dyson
Number of years of programming experience	8
Number of years of programming experience in asset's language	.5 in Ada
Education (list degrees)	BS Civil Engineering
Experience with Certification Tools (hours with each tool before starting certification process)	
Borland C++ IDE	0 hrs
PC-Lint	0 hrs
McCabe Visual Toolset	0 hrs
C-Vision	0 hrs

ASSET DESCRIPTION WORKSHEET

ASSET NAME	Labels
Origin of asset	Borland International
Application domain	Information Management
Purpose of asset	Updates and displays the contents of a mailing list.
Language	C++
Number distinct "includes" contained in the asset	5
Physical lines of code *includes blank lines and comments*	4828
Source lines of code (physical) *includes non-blank, non-comment lines*	3356 (est.)
Age of asset	1993
Version number of asset	1.0
Previous inspection and testing activities	unknown
Additional documentation	short prologue

Effort

Effort to apply the techniques for each step of the certification process was reported on the Overall Process Data Worksheet. Included in the reported effort is the effort to record defects, but not the effort learn how to use the tool. The graph in Figure 4-3 compares the actual effort to apply the techniques to the predicted, or default, effort. Default effort data is taken from CRC's Volume 3-Cost Benefit Plan.

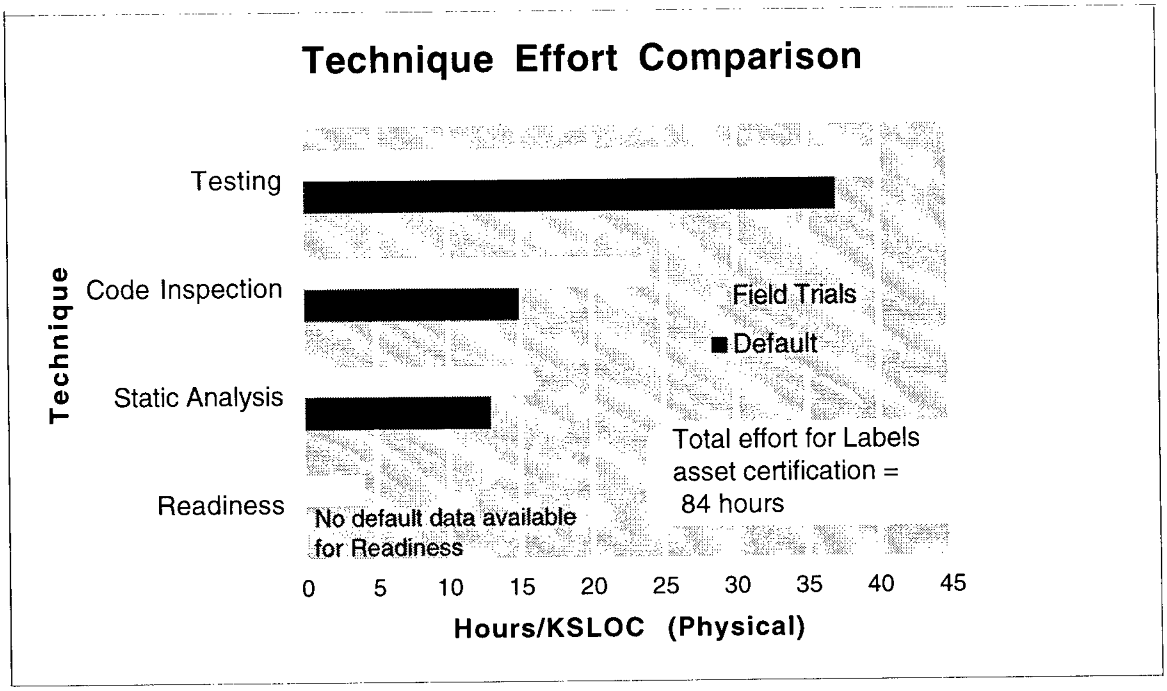


Figure 4-3. Comparison of Actual Effort to Predicted

In general, the actual effort was close to the prediction.

Since our initial effort of structural testing yielded high coverage (i.e., 97%), we elected to conclude the testing activity.

OVERALL PROCESS DATA WORKSHEET

ASSET:	Certification Step			
Labels	ASSET READINESS	STATIC ANALYSIS	CODE INSPECTION	TESTING
Certifier ID	Sharon	Sharon	Sharon	Sharon
Level of Effort (hrs)	4	16	24	40
Problems in Applying Techniques				
Problems in Using Tools	Borland required proper path settings for all included libraries and supporting reference files			Borland 5.00 and McCabe 5.2 were incompatible; upgraded to 5.01 and 5.22, respectively
Problems with Process Guidance				
Other Problems				

Defects

Many more natural defects were found in the asset during the field trial than were known prior to the start. All are recorded on defect report forms in Volume 2, Additional Certification Field Trial, Appendix B. Each report has an identifier that indicates the source of the report using the following codes.

Defect Report Identifier Codes

Code	Source
RD	Readiness
SA	Static Analysis
CI	Code Inspection
TE	Testing
PA	Aymond's Seeded Defect

In terms of certification, the asset passed the certification concern of Completeness, and failed in the other two concerns of Correctness and Understandability. In practice, the certifier would face the following choices:

- Reject the asset
- Report the asset as uncertified and record all known defects
- Return the asset to the donor and request repair of known defects; repeat the certification process after repairs
- Repair the defects; repeat the certification process after repairs

Some certifiers may choose to include defect repair as part of their certification process. There is some debate as to whether it would be necessary to repeat the certification process after repairs have been effected, depending on the nature and the number of the defects found. The purpose of repeating the certification would not only be to insure that the defects were repaired, but also to catch any new defects inserted as a result of the repair activity.

Counting Defects. In the following graphs and tables, unless otherwise noted, defects are counted as unique defect reports. The uniqueness criterion means that if the same defect was detected by more than one technique, it is counted only once and credited to the first technique to detect it. In filling out the defect reports, each report is limited to a single package or separately compilable file. All occurrences of the same type of error, such as a style violation, in a module are recorded on the same report, with all defective lines of code noted on the form.

Figure 4-4 shows how many defects were found by the steps in the certification process versus how many are known to exist at completion of the field trial. Defects categorized as *not found* are seeded defects.

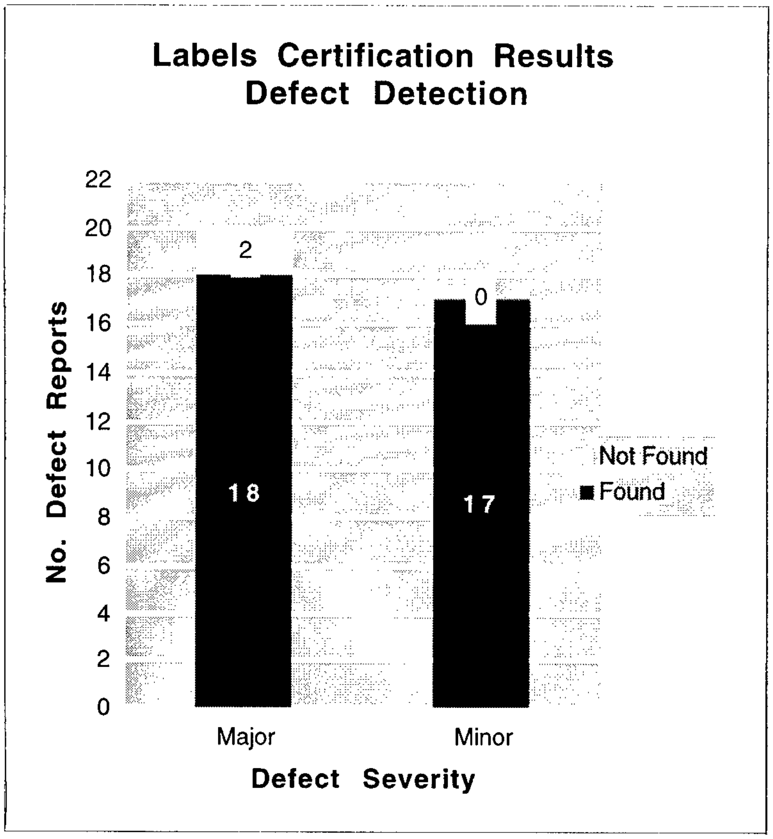


Figure 4-4. Defect Detection.

Summary of Defect Reports. The following table summarizes the defect reports logged during the certification process steps and the seeding activity. Duplicate reports are listed in the "prior step" shaded rows.

Defect Report Summary

		Defect Type					
Step	When Found	Comp.	Data	I/F	Logic	Other	Total
Readiness	This Step First	0	0	2	0	0	2
Static Analysis	This Step First	0	3	8	3	0	14
Code Inspection	This Step First	0	0	2	6	1	9
	Prior Step			1	0		1
Testing	This Step First	0	0	0	9	1	10
	Prior Step				0		
Seeding	Not Found	1	1	0	0	0	2
	Other Steps	0	0	4	8	0	12

Asset's Defect Profile. Figure 4-5 shows the defect profile of the asset in terms of the known defects. The defect density of the asset's major defects, including the seeded defects, is about average for C [see CRC's Cost Benefit Plan]. Major defects as we've defined them for the field trial are equivalent to what are typically reported as defects.

Defect Density

	Defect Density	
Defect	(defects/1000 physical lines)	
Severity	Asset's	Average for C
Major	6	6
Minor	5	N/A

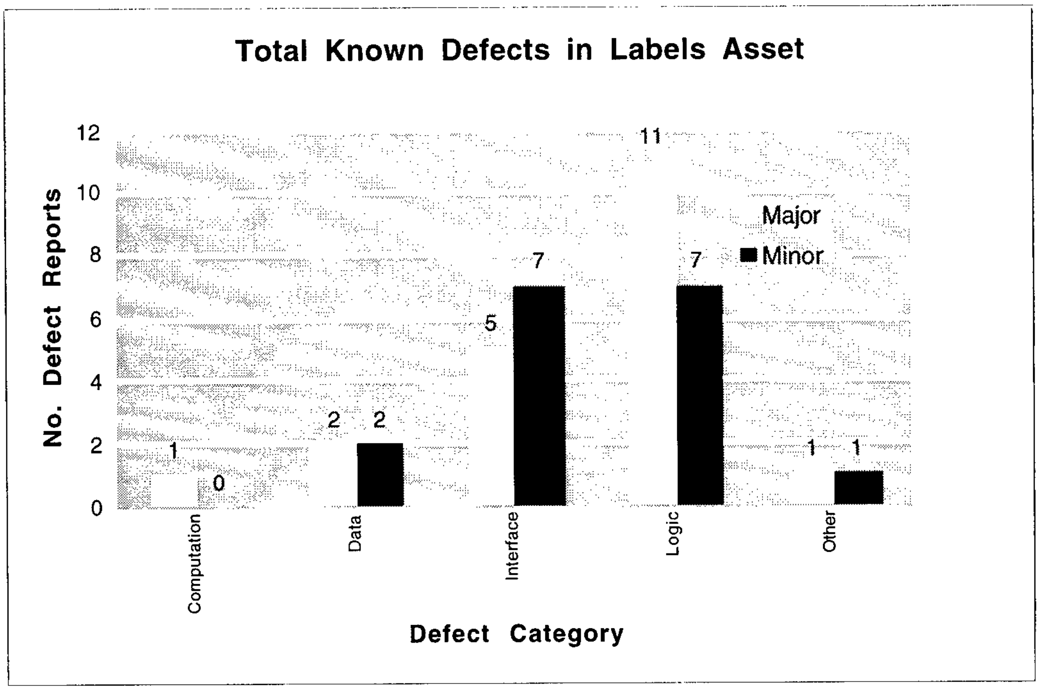


Figure 4-5. Asset's Defect Profile.

Figure 4-6 compares the asset's defect profile, including both major and minor, seeded and natural defects, to the default profile [see CRC's Cost Benefit Plan]. One notable difference is that there is a much lower proportion of computational defects. This fact could have two interpretations:

- the techniques used are not effective at finding computational defects
- the asset does not have computational defects

The second explanation is more likely, since the asset is not heavily computational in nature, only the date is computed in the labels program. No seeded defects were of the computational category. This then indicates that we cannot assess the effectiveness of the techniques at finding computational defects based on this field trial.

In certification, it will typically be the case that an individual asset's defect profile is different from the default profile of any given group of assets. The more that is known about the expected defect profile of assets to be certified, the more cost effective a process can be designed to certify them. For example, if a group of assets to be certified is known not to be computational, then you would not need to include a technique that is effective at detecting computational defects.

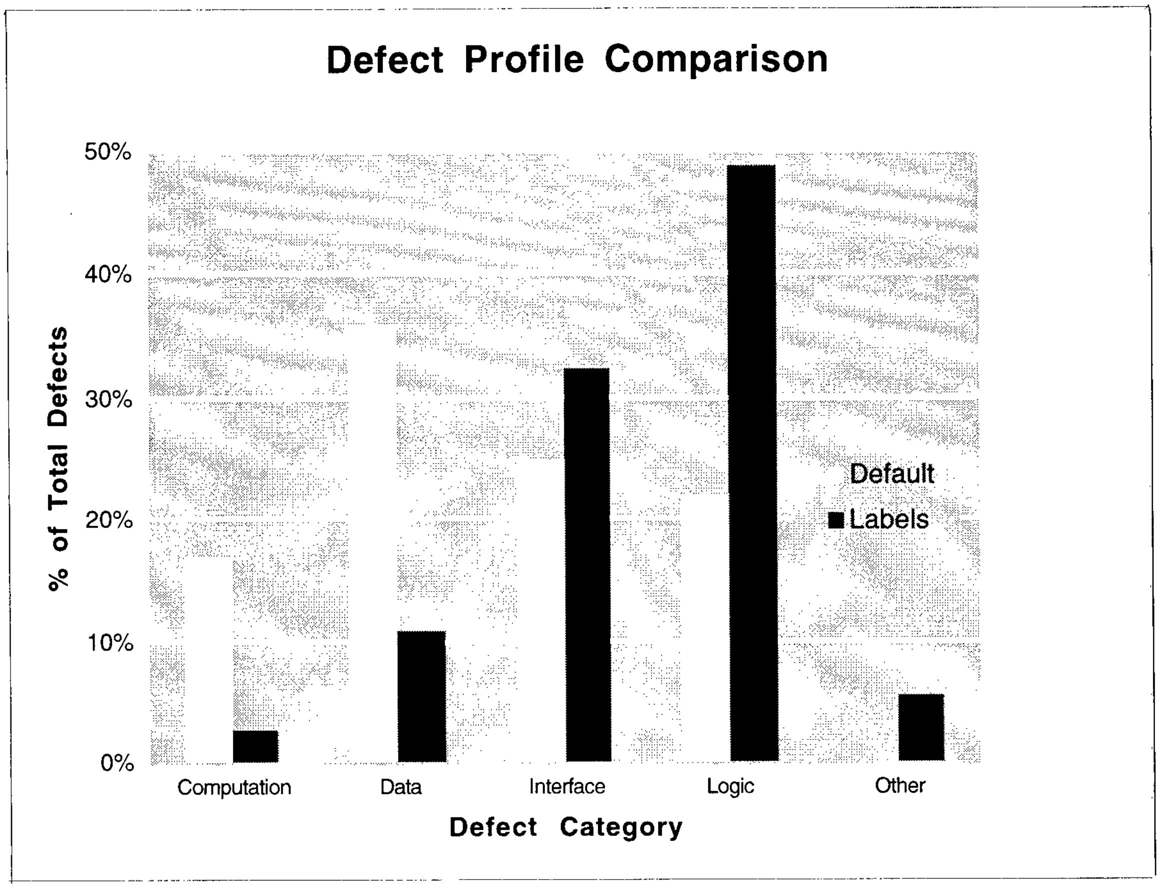


Figure 4-6. Comparison of Asset's Defect Profile to Default Profile.

Technique Effectiveness

As Figure 4-4 shows, all but two of the known major defects was found, and the two not found were seeded defects. Effectiveness of the default certification process at finding defects is better represented by the proportion of the total seeded defects found than by the proportion of known defects found. This is because there may be additional natural major defects in the asset, so the total number defects in the asset is unknown.

Effectiveness at Detecting Major Seeded Defects

Found	Known	Effectiveness
18	20	90%

Figure 4-6 shows the cumulative effectiveness of the steps in the certification process where effectiveness is defined as the proportion of known defects found. From this we can draw several important conclusions. We cannot, however, claim that the combined effectiveness of the default certification process is more than 90% because we do not know the total number of natural defects in the asset. Furthermore, based on the effectiveness at finding seeded defects, we have reason to believe that more natural defects exist.

Readiness step. There were two major defects found during the Readiness step. Even though initially, all code needed to create an executable was available and compiled without error, we found a major defect in documentation of the code's functionality. After we upgraded our Borland compiler to operate with the upgraded McCabe Visual Toolset, we uncovered a seeded error during linking in compilation.

Static Analysis step. As Figure 4-7 shows, both major and minor defects were found by this step. The particular tool selected for this step was very good at finding defects. The 55% effectiveness rating for minor defects shown on the graph may be misleading, however. The automated tools used in this step are virtually 100% effective at finding the defects that they are *designed* to find. The effectiveness rating indicates that what the tools are designed to find were only about half of the known minor defects in the asset.

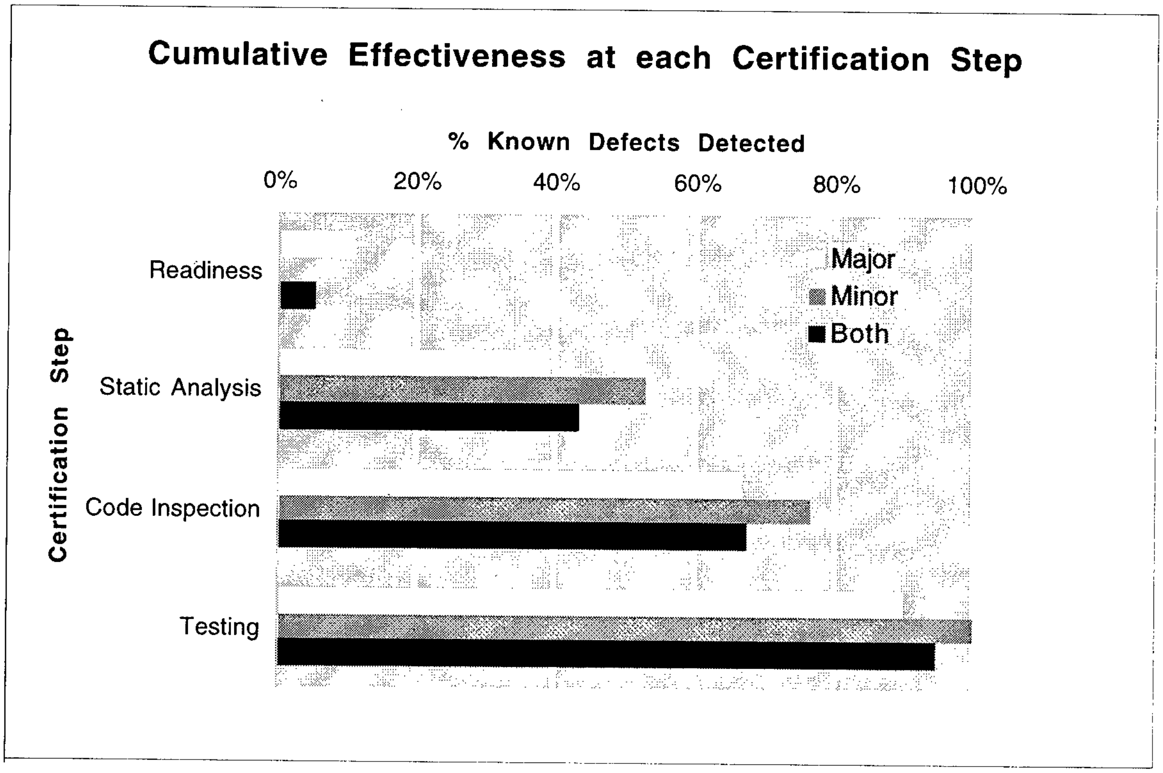


Figure 4-7. Cumulative Effectiveness of Certification Steps.

Code Inspection step. As Figure 4-7 shows, this step found about 25% of the major errors. This is lower than the industry studies that support code inspection as a useful technique to detect defects. The first field trial also had a lower than expected result. Consequently, we modified our checklist to add additional granularity to the questions in hope of improving our results. Our repeated results show that this may not be the factor behind the shortfall. Other explanations may be the certifier skill and years of experience with the code asset language.

Testing step. Less than one-third of the defects were found in the testing step, as can be seen by subtracting the effectiveness of the code inspection step from that of the testing step in Figure 4-7. This may be low for this step, but using the cumulative effectiveness of other steps, adequate coverage was achieved.

Lessons Learned from the C++ Field Trial

Choice of Component Language

Even though C++ is a popular and industry-endorsed language, several flavors are in existence. These are two standard forms (i.e., ANSI/ISO and ARM), but others have created de facto standards. These varieties come into play when choosing compilers and tools that pre-process code. Different flavors of the C++ language pose interoperability problems. Some tool vendors do not support a wide variety of C++ flavors and special customizations of the tool need to be performed. These customizations are not supported by the tool vendor. These factors eventually affected the selection of the asset to be certified.

Tools that support C++ are not robust. C++ is widely acclaimed as an excellent language of choice over C, but this trend is a fairly new one. Tool vendors need additional time to provide mature tools to meet the market demand.

Defects

All defects found in the Testing Step were unique. The first field trial has some minor overlap of errors found in succeeding steps and separation was not as clearly evident as in the second field trial. Nonetheless, this finding confirms that a certification process should include a series of steps using distinct techniques designed to detect different kinds of errors. Overall, we found that each technique is special and cannot be omitted from the process.

The components used Field Trial #1 and Field Trial #2 differed in the total number of minor defects. Field Trial #1 found 77 of a total of 85 minor defects and Field Trial #2 found 17 of a total of 17. This may be due to the differences in the initial, unseeded component, as well as the differences in tools used in the two certification environments. Field Trial #1 had the advantage of AdaQuest to find minor violations of coding style whereas no such tool existed as a counterpart in the C++ certification environment. In Field Trial #2, PC-Lint was used as a thorough static analysis tool and can be thought of as a parallel tool that detects minor defects.

Many major defects were found in the earlier certification steps (i.e., prior to Code Inspection). This finding also confirms the need for a multi-step certification process.

Defects found in earlier steps are less costly to find and to repair than those found in later steps. Finding defects late in a development process (i.e., during testing) is not usually cost-effective.

Defect Categories

The categorization of defects, both seeded and natural, is difficult to assign from the definitions alone. The definitions as they appear in the CRC Code Defect Model could be improved by elaboration with additional details specific to each component language. Examples to illustrate assignments of categories would be helpful.

The Field Trial procedures would benefit by adding these examples for each kind of defect to help the Certifier and Certification Analyst to make this determination. We were able to adequately maintain consistency across the two Field Trials conducted at SPS through individual staffing.

Field Trial - Certification Tools

The configuration of the certification environment is time-consuming. We needed to artificially create the experimental environment prior to conducting the test. In a repository situation, this environment would already be established.

Installation, learning, integration, and application of tools to a particular component is very time-consuming. The activities are difficult to plan because of unknown obstacles that are encountered. It is suggested to build a three month period into the schedule for these activities alone. Using an example component that is available to the tool vendor's technical support staff is helpful in tracking bugs and errors in installation and operation of the tool.

Configuration and integration of tools is problem-fraught. Version incompatibility across tools can present problems in operation. Tools are marketed as compatible, but, as each vendor may issue monthly changes, particular versions of one tool may not work with a version of another. Upgrades to one tool may cause an new incompatibility in another tool which once functioned properly. Fortunately, for Field Trial #2, vendor support was excellent and enabled us to work through the barriers.

Since vendors issue frequent versions of their software, documentation does not match tool versions. Patches may be available, but are difficult to secure. Installation of patches may be time-consuming and problem-ridden. This presents problems with those who are learning the new tools or learning the differences in the new version.

Support for tools that instrument code is weak. For example, the instrumentation mode was not sufficiently tested using a sample program provided by the vendor with the Borland compiler and McCabe Visual Toolset. Documentation of the process was non-existent and was created "on the fly" as the problem was solved. Bugs in the tools were uncovered as the problem was resolved. We recommend that tool vendors who have an instrumentation mode provide samples to test tool installation and functionality.

With the McCabe Battlemap, the ability to jump to the actual source line of code from the Battlemap would improve its capabilities.

Training is a requirement for high-end tools. Complex tools give sophisticated results and require a high learning curve to operate the tools properly. User documentation is typically weak; we found this to be true of both Logiscope used in the first field trial and the McCabe Visual Toolset used in the second field trial. We found that McCabe provides manuals in large binders making it difficult to find the desired information. On many occasions, once the information was found, it was incorrect and out-of-date, not matching the most current version of the tool issued. Additional expertise is required to sift through the volume of information available from the tool and interpret the results. A high level of expertise is required to learn the tools, get them up and running, use, interoperate, and interpret the results.

Training for the McCabe tools focuses on the theoretical underpinnings of the tool's complexity measures and control flow theory. We found this useful; however, another course targeting the application of the tools to a real-world situation is needed. Currently, these services are available only on an in-house consulting basis and can prove to be very costly for those on limited funds.

For complex tools, an excellent technical support staff relationship is required. The tool vendors must be responsive to tool problems, otherwise, a failure to complete could result.

Field Trial - Testing Effort

The design of the component under test greatly affects the testing effort when using a structural testing approach. The component for Field Trial #2 had a flat calling tree structure and was highly coupled across modules. Modules were small and had low control flow complexity. This structure is typical, and can be expected, for a component implemented in the C++ language. Branch coverage of 97% was easily achieved. Whereas the calling tree structure of the component in Field Trial #1 was deeper and the modules were longer and more complex, it proved difficult to achieve more than 80% branch coverage.

Certifier Skills

The suite of certification techniques that comprise the default certification process includes two techniques whose effectiveness is highly dependent upon the training and experience of the certification engineer applying the technique: code inspection and testing. These techniques are also less automated and require more human involvement than the readiness and static analysis steps. This implies that the results may not be repeatable when comparing different certification engineers. To reduce the variability among different engineers, and to maximize the effectiveness of the techniques, training is essential.

The default process steps are intentionally ordered in terms of increasing skill level as well as increasing investment of effort, so that, for example, a failure in an early step could save wasted effort in later steps. In general, we would like the automated static analysis tools to detect as much as possible, and we view enhancements in static analysis capabilities as a valuable contribution to certification.

Effectiveness of Techniques

The combined effectiveness of all of the steps in the certification process is impressive because each step tends to find different types of defects. The second field trial confirms the results of the first, that all four steps are necessary to detect a high proportion of defects.

Figure 4-7 shows, for example, that many of the major defects would have been missed if we had only done static analysis. The Defect Report Summary table also shows that there are numerous defects that testing alone would not have found.

We recommend that defect detection be pushed to the earlier certification steps. For example, automated static analysis is a cost-effective, objective, non-cognitive technique as compared with code inspection which requires trained staff and considerable effort. The effectiveness of some techniques are contingent upon the persons using them.

The Code Inspection Step for Field Trial #1 and #2 were only moderately effective in detecting defects (i.e., 37% and 27% respectively). This may be due to a relatively small body of detectable defects over both field trials. A more definitive trial of the process would to certify multiple assets with thousands of defects. Here, in this experiment, we inserted "controlled" defects which may not necessarily be typical of the kind of defects that arise naturally.

We were impressed with the ability of the upgraded Borland compiler to detect a previously undetected major error during the Readiness step. We hope that this finding is a trend among vendor upgrades as support the software developer and maintainer in detecting defects early in the software life cycle.

Modifications to the Process Guidance

General. The certification process as defined by the steps of Readiness, Static Analysis, Code Inspection, and Testing is valid. Many natural defects, as well as the seeded defects, were found in the certified COTS components. Field Trial #2 found 7 natural defects, and Field Trial #1 found 12.

Code Inspection step. In C++ with numerous, short modules, code design and its "checklist" may become more important to major and minor errors, corrections understandability. Design appears more closely tied to implementation of function. It may be useful to add a reverse engineering tool to the certification environment to help understand code structure. We found that McCabe Visual Toolset does not provide sufficient insight.

Recommendations

Seeding defects was a difficult activity, and we cannot confirm that the defects seeded are typical of the defects that software developers and maintainers inadvertently introduce into source code. We recommend conducting a study to determine examples of defects that are typical across defect types.

Additional planned empirical research should attempt to validate the certification reuse process and procedures. Additional data could be collected for Ada, C++, components as well as other programming languages (i.e., COBOL, FORTRAN, Pascal, C, etc.) in follow-on pilot studies.

After a significant number of pilot tests, we recommend an additional phase of applying the certification reuse process to multiple components of a reuse library and collecting additional data analyses, and results for the purpose of comparison. The next phase of validation could involve multiple reuse libraries to determine the relative efficiency of those processes and procedures. The certification process could alternately be expanded to other quality concerns, other domains, and other component types.

The disappointing results achieved in the Code Inspection step, suggest a topic for future research, i.e., the study of ways to make code inspection more effective. This research topic is also of interest to software maintainers who routinely struggle with the comprehension of code written by others.

The results of the Field Trials is of interest to the software/systems community. The technical paper and presentation of the first field trial at the IEEE International Conference on Engineering of Complex Computer Systems '96 (ICES) was well-received and drew additional conversation from its participants. We intend to follow-up with an additional paper about the second field trial and its comparison to the first at a future conference.

4.1.2 Support of Pilot Sites

In addition to the CRC research team consisting of SPS and its subcontractors Veriquest and GRCI, two other organizations interested in certification reviewed the Certification Framework and certification process for code assets. These organizations are considered pilot sites for the CRC certification technology. Activities undertaken as part of this effort to support these pilot sites are documented in this section.

The first pilot site organization was the Air Force Reuse Center at Maxwell AFB Gunter Annex including Gunter personnel supporting the Base Level Systems Modernization (BLSM) I program. The Reuse Center is concerned with certification of reusable components, and expressed an interest in the certification tools and techniques. Gunter personnel reviewed CRC documents and were briefed on the results of the first certification field trial. To illustrate how the CRC cost/benefit model may be customized to suit a particular organization, SPS analyzed a set of problem reports from the Sustaining Base Information Services (SBIS) components developed under BLSM I. The results of the analysis are presented in section 4.1.2.1. Gunter personnel were briefed on the results of this analysis.

The second pilot site organization was Underwriters' Laboratory (UL). UL is a commercial organization that certifies the quality and safety of many products. Now that software has become a larger part such products, UL is in the process of developing a certification approach that encompasses software. UL has reviewed the CRC Certification Framework and related documents and has provided review comments during the course of the CRC effort and this effort. Under this effort, the subcontractor GRCI provided assistance in acquiring and installing the certification tools at UL. The results of UL's review of the Certification Framework and UL's field trial experiment were documented under a separately funded effort.

4.1.2.1 BLSM I Problem Report Analysis

SPS analyzed a set of 47 problem reports from the BLSM I program in order to extract the data necessary to customize the CRC cost/benefit model to the Reuse Center organization. This customization approach is based on the premise that past performance is a predictor of the expected types of defects, and the rework effort associated with these defects, for future reusable components. Even though the specific components represented in the problem reports are not expected to be reused in a future development effort, the data derived from the problem reports is expected to be a better characterization of the organization than the generic (default) data presented in the CRC Cost/Benefit Plan document.

The objectives of the analysis were to develop a custom defect profile and a custom effort profile based on the problem reports. The first step was to assess the set of problem reports to determine its size and scope. This analysis assumes that SPS received all available problem reports for the CUB, AFORMS, MDS, and LOGMOD B components. In the absence of a complete set of problem reports, it would necessary to have a representative sample, and the required sample size would depend on the total size of the population.

By plotting the problem report control number (assumed to be a unique identifier) versus origination date, we can see the pattern that emerges as shown in Figure 4-8. As a whole, the set of problem reports covers roughly a three year period. The Common Utility and Bindings (CUB) components problem reports cover the longest period of time. These CUB components were used in building the other systems (AFORMS, MDS, and LOGMOD B). The CUB pattern shows that some CUB problems were detected prior to the creation of the other 3 systems (pre-integration), but most were found post-integration.

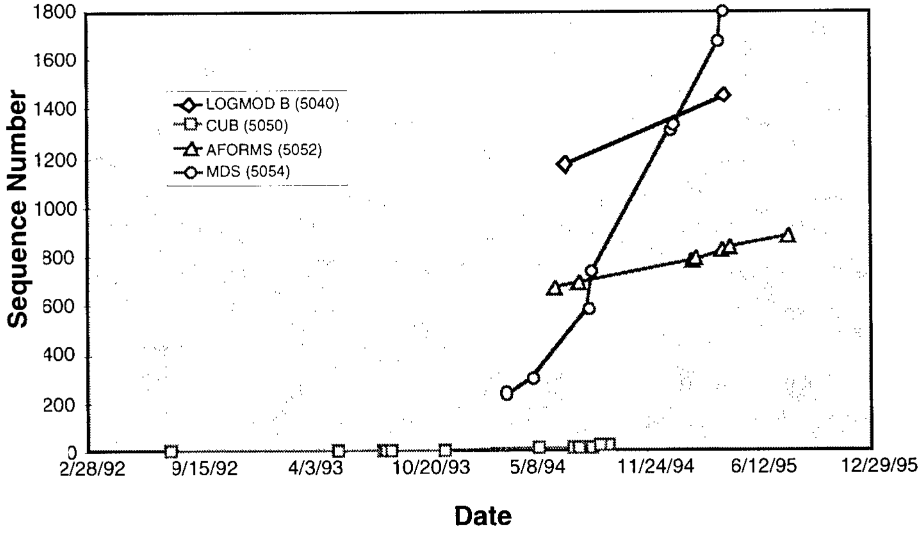


Figure 4-8. Control Number Characterization

The pareto analysis in Figure 4-9 shows that the largest number of problem reports in the sample belongs to CUB components. Pareto analyses, a special form of a vertical bar graph are useful in determining which problems to solve in what order. We will generally gain more by working on the tallest bar than tackling the smaller bars. A line can be added to show the cumulative frequency of the categories and answers the question, "How much of the total is accounted for by each category."

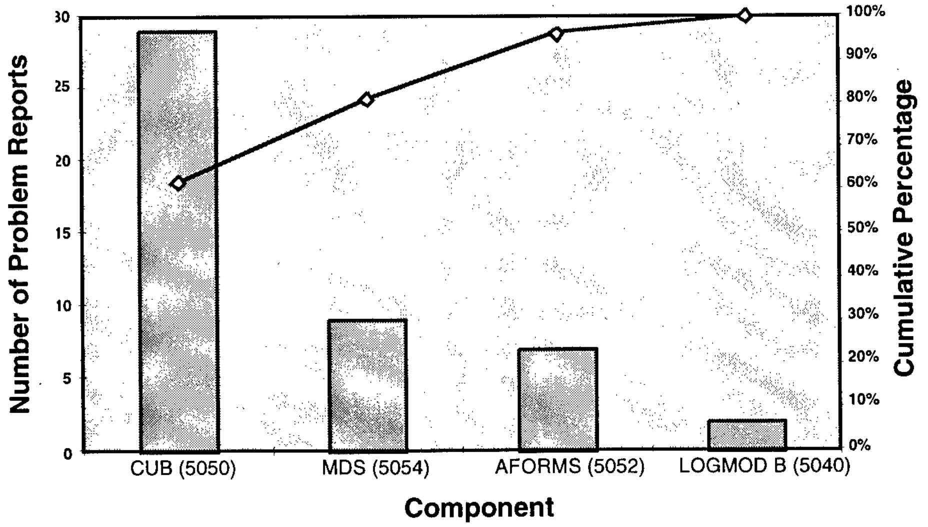


Figure 4-9. Pareto Analysis of Problem Reports

Defect Profile Determination

To construct a defect profile, each problem report was categorized in one of the five defect types of the CRC defect model (computational, data, interface, logic, and other) based on the information recorded in the report. The results are shown in the pareto diagram in Figure 4-10. The implication of this profile for certification is as follows: the large percentage of logic errors suggests that testing should be part of the certification process.

The large portion of defects in the "other" category suggests that the categorization scheme could be improved for this organization. The "other" category should be examined for natural groupings and another defect type or types should be created. For example, reports of type "other" may be associated with non-software problems, and thus perhaps should be excluded from analyses.

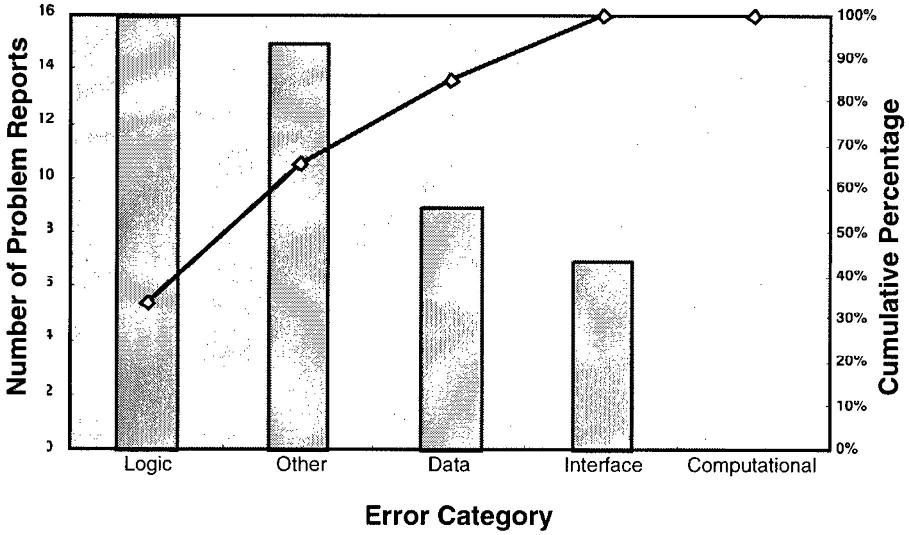


Figure 4-10. Error Category Pareto Analysis

By breaking the defect profile down by system (component groups) as shown in Figure 4-11, we can see that the profile (i.e., the proportion of each defect type) varies by system. Because the other systems have significantly less reports than the CUB components, there is insufficient data to construct system-specific profiles. Therefore, the overall profile shown in Figure 4-10 is the best characterization of the components as a whole.

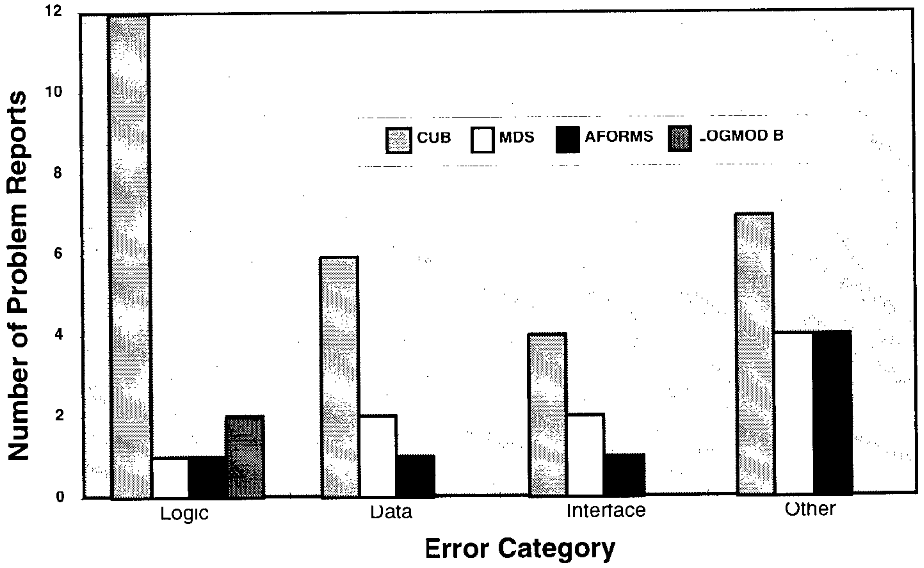


Figure 4-11. Distribution of Error Categories

Effort Profile Determination

Rework effort for each problem report includes the effort to analyze the report plus the actual hours to repair it. Not all reports have reported effort data. Figures 4-12 and 4-13 show the portion of problem reports that have effort data.

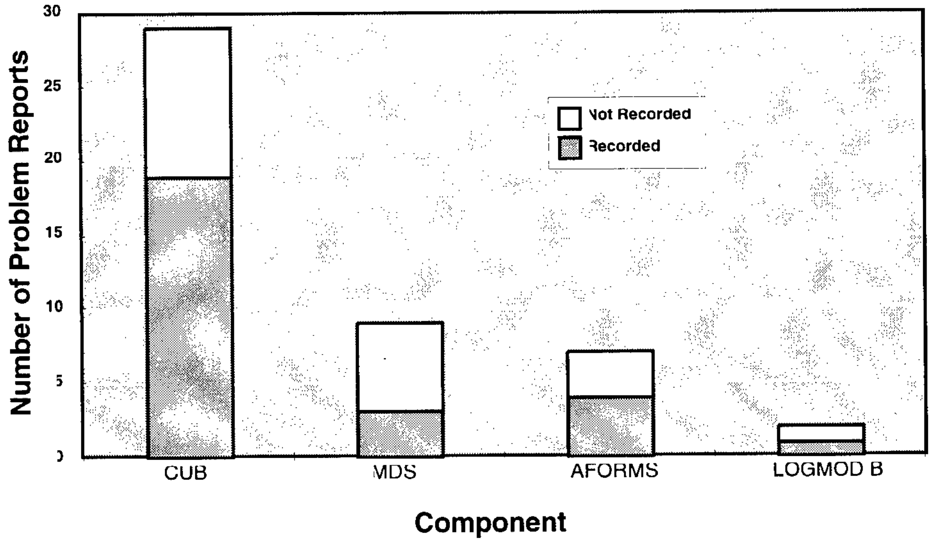


Figure 4-12. Recording of Analysis Hours

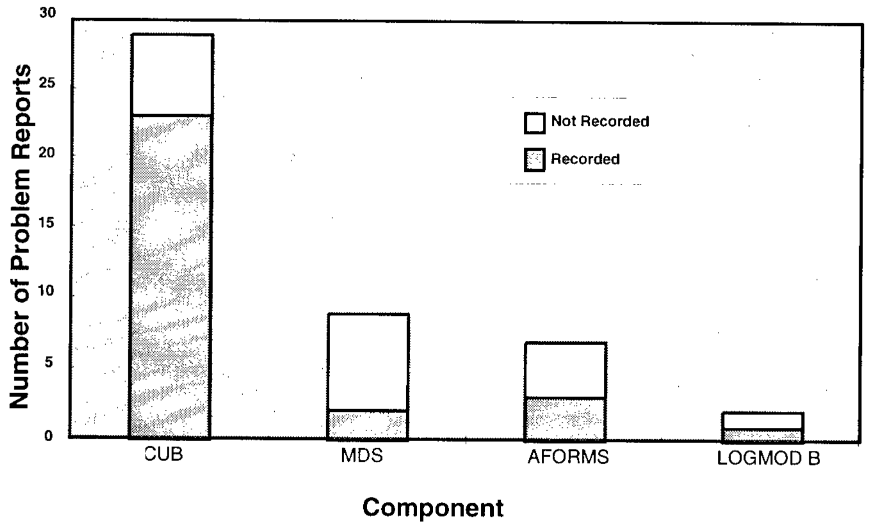


Figure 4-13. Recording of Actual Hours

Because the CUB components problem reports was the largest group and also had the largest portion of effort data reported, the rework effort analysis was limited to the CUB component problem reports. The effort data was plotted in a histogram showing the number of reports with recorded values in the range (or interval) shown on the x-axis. For example, Figure 4-14 indicates that 8 reports had analysis effort of less than or equal to one hour. Figure 4-15 shows a histogram of the actual hours effort data. However, since this analysis is based on approximately 20 problem reports, it is not expected to be an accurate predictor of rework effort for future problem reports. The sample size is also too small to compute an average rework effort for each defect type. A more complete set of effort data is required; this analysis should be repeated after all of the problem reports have been resolved.

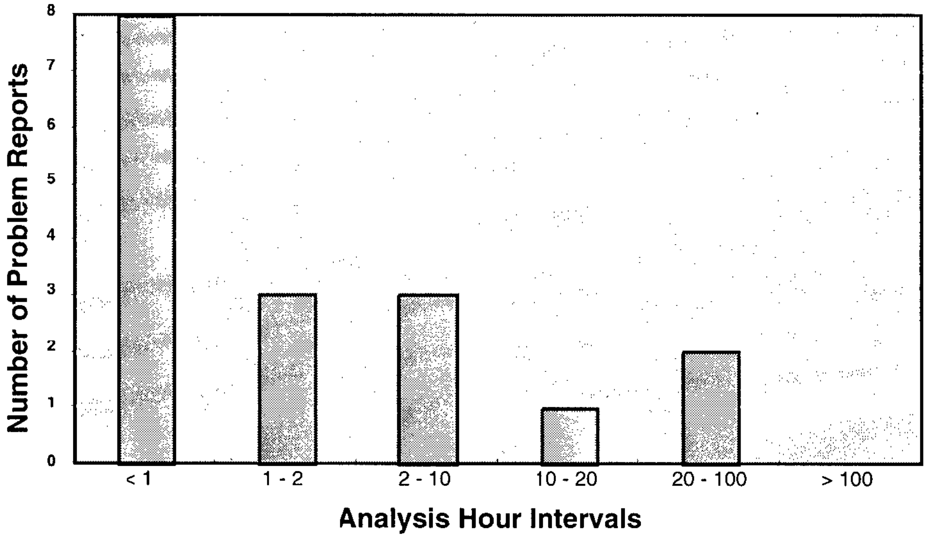


Figure 4-14. Analysis Hours for CUB Components

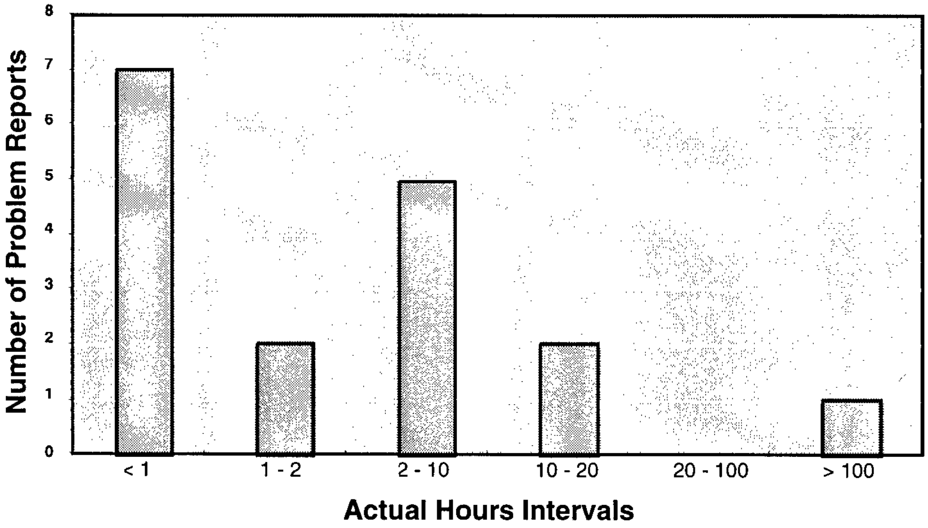


Figure 4-15. Actual Hours for CUB Components

The effort values in the histograms of Figures 4-14 and 4-15 can be converted into an rework estimation value using the method shown in Table 4-1. The average number of hours for an interval is multiplied by the proportion of reports in that interval to compute the contribution of that group of reports to the estimated average value. This method provides a better estimate than a simple average calculation. If more complete effort data becomes available, this computation can be repeated to derive a more accurate prediction.

Table 4-1. Estimating Rework Effort from Figures 4-14 and 4-15

Average Hrs in Interval	Proportion of Reports	Contribution
	Analysis Hours	
0.5	x 8/17	= 4.0
1.5	x 3/17	= 0.3
6	x 3/17	= 1.1
15	x 1/17	= 0.9
60	x 2/17	= 7.1
100	x 0/17	= 0.0
	Est. Average	= 13.4
	Actual Hours	
0.5	x 7/17	= 0.2
1.5	x 2/17	= 0.2
6	x 5/17	= 1.8
15	x 2/17	= 1.8
60	x 0/17	= 0.0
100	x 1/17	= 5.9
	Est. Average	= 9.9
Total Estimate	= 13.4 + 9.9	= 22.3 hrs

Other Observations

In addition to the above analyses aimed at generating custom defect and effort profiles, the following analyses were also performed to further characterize the set of problem reports.

- Severity analysis
- Category analysis.

The pareto diagram in Figure 4-16 shows the distribution of the severity classifications of the problem reports. Critical and Significant problems account for more than half of the problems, which is not typical of severity distributions for software that has been fielded. Generally, one would expect these severity categories to be a smaller proportion of the total.

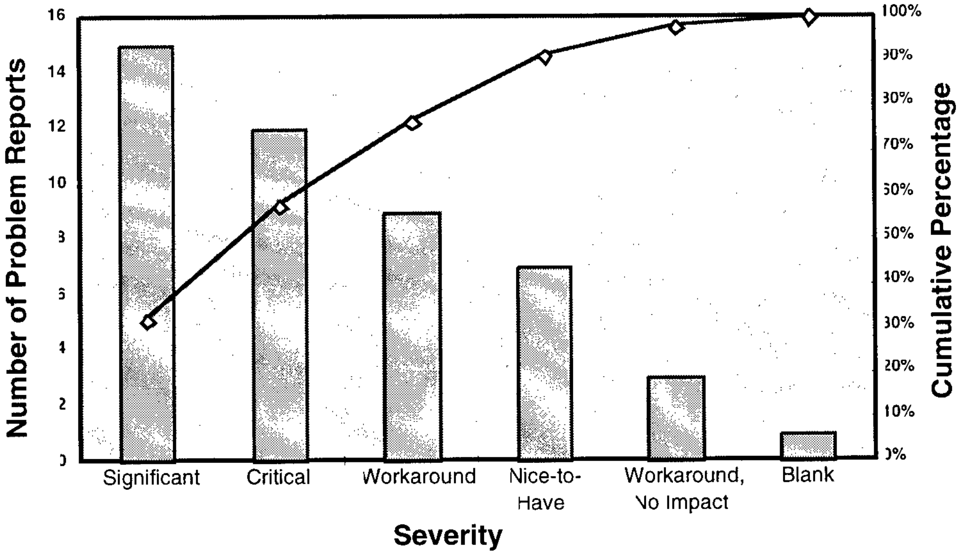


Figure 4-16. Severity Pareto Analysis

By looking at the breakdown by system in Figure 4-17, we find that the CUB components are principal source of critical and significant errors. Rather than indicating relative quality of the components, this may indicate that the CUB components have been more heavily used and/or tested.

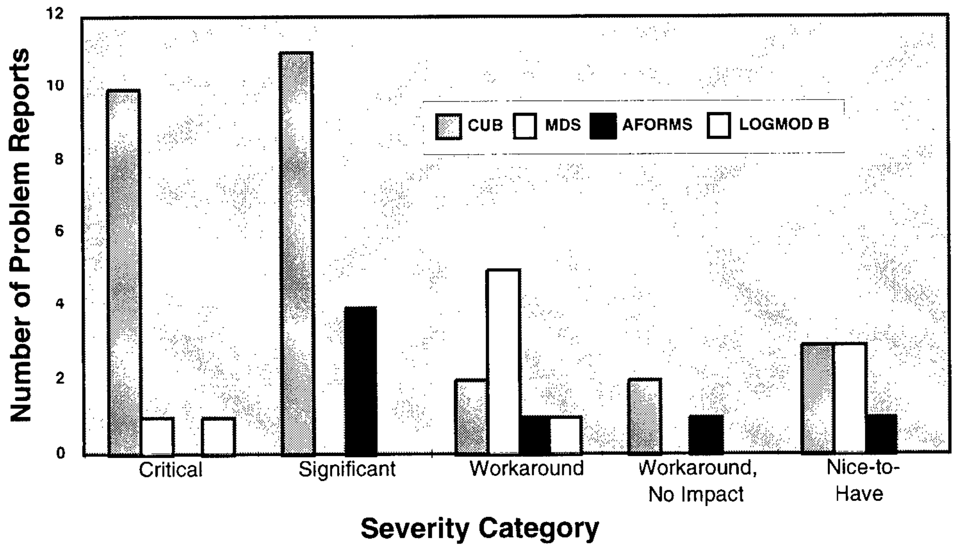
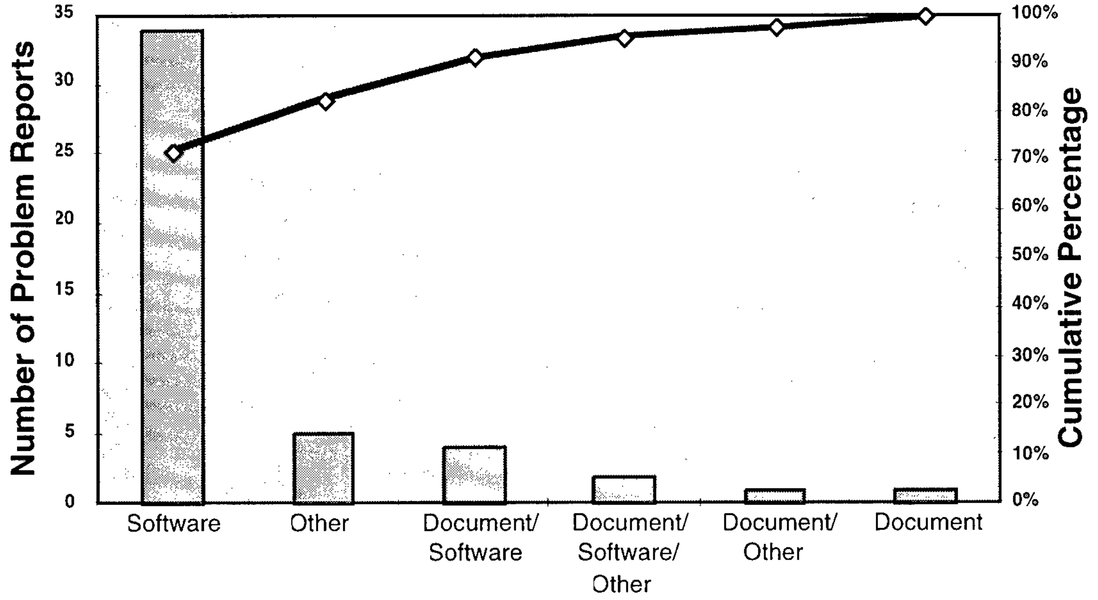


Figure 4-17. Distribution of Severity Categories

As Figure 4-18 shows, software problems are by far the most prominent category (or source) of problems. This is a typical pattern.



Problem Category

Figure 4-18. Problem Category Pareto Analysis

Analysis Summary

We were able to construct a custom defect profile for these components as shown in Figure 4-11. However, the set of problem reports with recorded effort data is too small to construct a rework effort profile. The pilot site may choose to repeat the analysis after the problem reports have been resolved and effort reported.

4.2 Certification Framework's Applicability to Commercial Organizations

In an effort to expand the Certification Framework's applicability to commercial organizations, SPS performed the following two tasks:

- Supported industry standards efforts in reuse and certification through the Reuse Interoperability Group (RIG)
- Conducted a survey of commercial organizations to determine how reuse and certification is being applied

The results of both of these tasks are discussed in the subsections that follow.

4.2.1 Commercial Standards Organizations-RIG

Advanced thinking on the part of the Advanced Research Projects Agency (ARPA), STARS, and the U.S. Air Force Reusable Ada Avionics Software Packages (RAASP) led to the creation of the Reuse Interoperability Group (RIG) in 1991. Participants in the

RIG represent several commercial organizations (e.g., Raytheon, MCI, Morgan Stanley, and NASA) who potentially adopt the RIG's work products.

As part of the RIG, its Technical Committee 4 (TC4) is one of the few groups addressing reuse and certification standardization and operational policies. It was supported by CRC and ATD to increase our awareness of the group's activities, as well as document their charter and progress. Observations during our participation in the TC4 confirms our evaluation of the state-of-the-art of reuse and certification and has helped to flavor our projects' efforts. Participation in activities such as these provided an opportunity to direct and guide the maturation of reuse and certification standardization.

The TC4 identified areas where standards for asset evaluation and certification will promote reuse library interoperability. The RIG standards are created using procedures defined by the Institute of Electrical and Electronics Engineers (IEEE). These standards serve as policies for industry rather than providing guidance for internal certification processes.

The working group submitted 3 standards to IEEE for ballot. These are the P1420.2 Basic Interoperability for Data Models for Reuse Libraries, the P1420.1a Asset Certification Framework, and the P1430 Concept of Operations for Interoperating Reuse Libraries. Currently, P1420.1a, Asset Certification Framework has been accepted by IEEE as a standard and can be ordered from that agency.

Through our participation in the TC4, the CRC and ATD efforts have influenced the group's work products and coordinated our projects' approaches. The TC4's mission is at a higher level of abstraction with a broader context; the CRC Certification Framework can provide a more detailed view of the issues in reuse and certification. Support by CRC and ATD helped us stay abreast of the TC4's activities, augment their knowledge and continue to influence development in the areas of reuse and certification.

4.2.2 Survey of Commercial Organizations

The purpose of surveying commercial organizations was to determine how certification is being applied. Our goal was to identify their reuse methods and the need for certification in the reuse process. The survey used an interview approach similar to that for the selected pilot site organizations in the Rome Laboratory Certification of Reusable Software Components project.

Candidates to survey were supplied internally by SPS. In pursuing reuse as a corporate business thrust, SPS has interacted directly with more than 700 companies, government organizations, and educational institutions interested in software reuse. Many of these interactions are documented within SPS' Telemarketing Database. This database was the primary source of candidates to interview as a survey of commercial organizations.

The objectives of the interview were to gather the following information:

- What kind of certification process is being used?
- What are the reuse goals?
- Is any part of the certification process automated?
- Where, in the software life cycle, is certification being performed?
- Who is doing the certification?
- What is the source of the assets to be certified?
- To what extent is certification being performed?
- How is the certification information relayed to the reusers?
- Who are the users of the certified assets?
- What incentive exists for the asset developer to develop reusable assets?
- Will certification increase the incentive for reusing assets?
- Does the participant perceive a benefit from asset certification?
- Would certification appeal to the reuser?
- What incentive exists for the reuser to reuse assets (i.e., does certification play a role)?

Using the SPS proprietary Telemarketing Database, we identified thirty organizations to interview which were actively performing reuse. Limiting our number of organizations to thirty provided us a statistically sound survey results and placed a reasonable limit on the survey effort.

Several types of survey techniques were investigated for gathering the needed information (i.e., telephone survey, mail surveys, on-site surveys, etc.). We chose telephone surveys since this technique promised the most cost-effective and highest rate of return. Mail surveys and on-site surveys proved unnecessary since telephone surveys provided adequate information. As an incentive for our survey participants, we offered to provide the results of our survey.

The survey process consisted of the following five phases as shown in Figure 4-19.

1. Prepare the Survey Material
2. Identify the Survey Participants
3. Perform the Survey
4. Analyze the Survey Data
5. Document Results

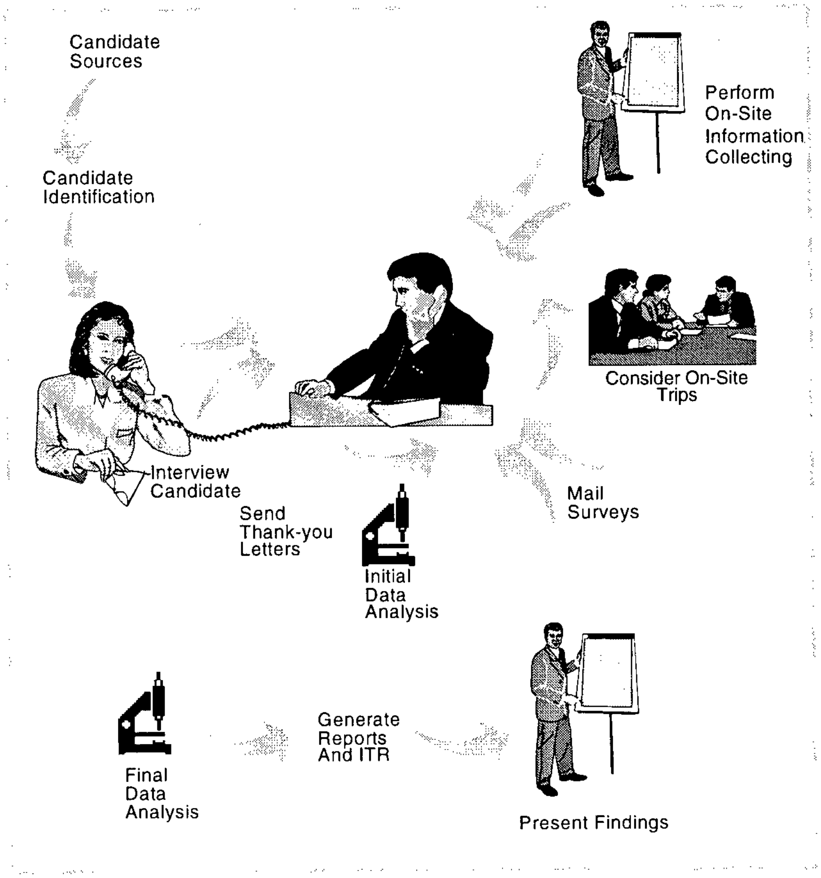


Figure 4-19. Survey Process

The data collected from each of the telephone surveys is captured in Appendix C - Survey Data. The following discussion summarizes our survey results.

During the survey, we interviewed, or attempted to interview, a total of 33 people. Seven people could not be contacted, and three refused to discuss any information about the reuse activities of their company. These participants were from a wide range of companies (e.g., DoD contractors, banks, credit institutions, and universities). All of the participants were project leaders or higher rank in their companies. This was due to the nature of SPS' Telemarketing Database since it is composed of people who are responsible for purchasing software for their group or company.

Nearly all of the participants were willing to provide information about their reuse efforts. Those that were hesitant became willing after they were informed of the intent and confidentiality of the survey.

Upon interviewing these individuals, it was observed that reuse was performed at an informal level. Twelve companies were in the process of establishing reuse systems or expanding systems already in use. Of these, a few were designing a very detailed reuse process as well as asset repositories that would store substantial asset information.

Of the fifteen participants performing reuse, or initiating a reuse system, nine had positive feedback towards asset certification. Participants in companies that were in the process of establishing elaborate reuse systems stated that they would not use a non-certified asset. Two contacts stated that an asset would not be considered reusable unless it was certified. Seven contacts were actively doing a level of asset certification.

Five contacts did not think that certification was a reuse issue although most of the five did think that certification was beneficial.

Nine of the seventeen contacts that were performing reuse stated that certification was a reuse incentive. Six did not think that certification was an incentive for reuse. These six had other priorities, such as understanding the asset and accessing the asset.

Overall, about 40% of our survey participants believed that certification was an incentive to reuse. About 20% did not see certification as a reuse issue. The remaining 40% had not thought about the certification issue enough to be able to comfortably respond to certification questions.

Based upon our overall survey results we believe that about 50% of commercial companies are involved in reuse. While most of these companies are just starting to reuse software, all have an informal reuse process in place.

This survey and its assessment, together with the State-of-the-Art Report on Reuse Libraries in Appendix D, lead us to the following conclusions. We believe the future reuse arena holds a place for certification tools; those contacts with well-defined reuse goals stated that certification is necessary for reuse. While reuse is still immature in the commercial market, it has supporters. Eighty-five percent of the survey participants are in the process of advancing their reuse process. Only about 20% of the twenty-two companies surveyed were associated with projects which did not require either reuse or certification.

The current reuse market is not advanced enough to be able to determine the kinds of certification tools that would aid in their reuse initiatives. Additional R&D is required to determine the necessary tools.

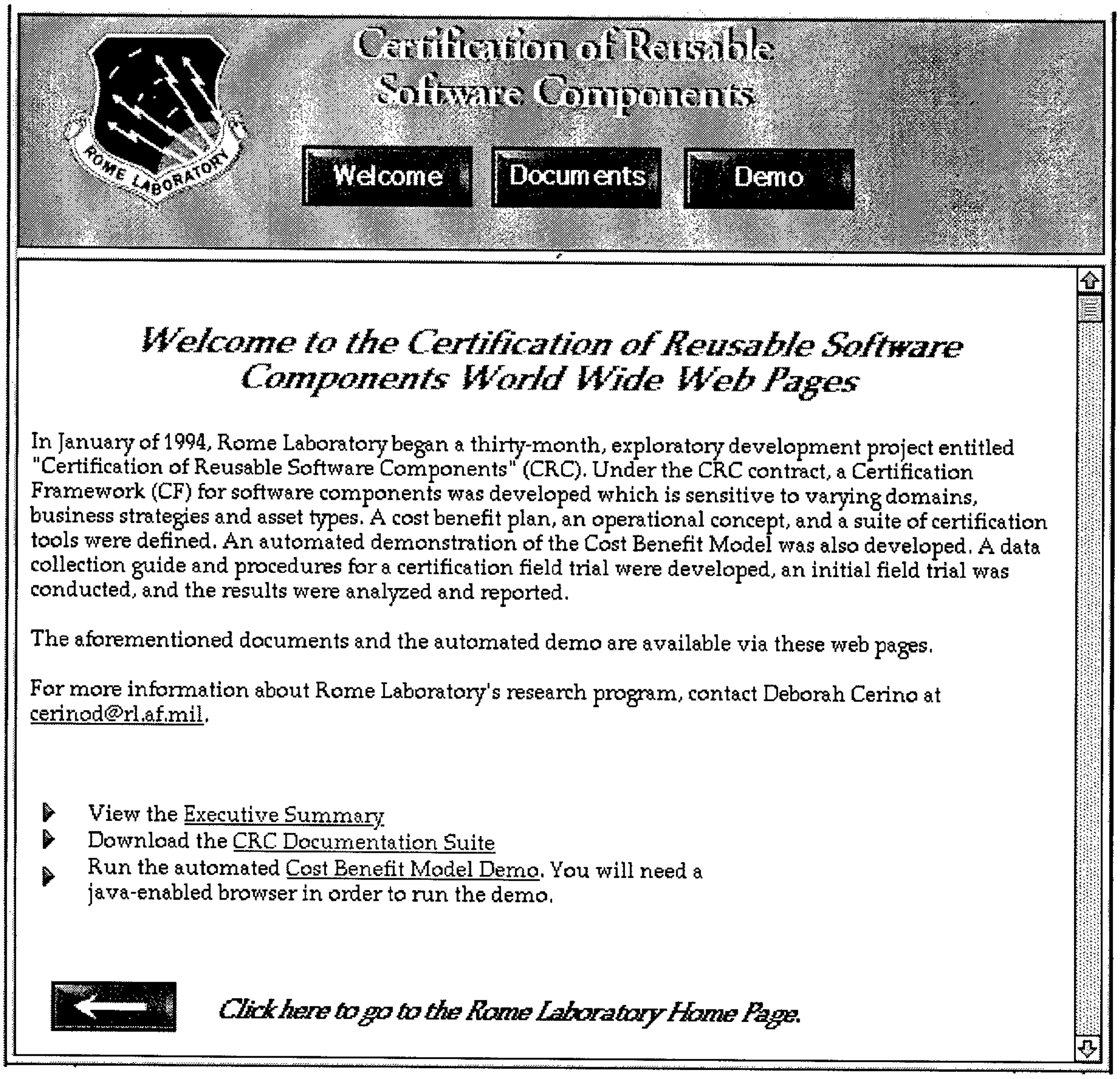
As a potential follow-up, the interview participants could benefit by the use of certification tools if the tools were perceived as a reuse aid. Such tools would enable companies beginning their reuse programs to advance more rapidly.

4.3 Certification of Reusable Components (CRC) Web Pages

Under this effort, the Wordwide Web (WWW) pages for the CRC effort were completed and were integrated with the automated demonstration developed by the subcontractor GRCI. The objective of the web pages is to provide information about this research initiative at to a wide audience. In addition, the web pages provide a convenient means for interested persons to acquire a copy of the CRC document suite. These web pages will be linked to the RL home page and hosted at the RL server.

The CRC web pages are written in hypertext markup language (HTML), which is the standard for documents that are viewed with web browsers such as Netscape and Microsoft's Internet Explorer. The automated demonstration is an interactive program that illustrates one of the CRC cost/benefit models in action. This demonstration is a Java applet, and requires a Java-enabled browser (such as Netscape 3.0 or Internet Explorer 3.0) in order to execute.

The CRC welcome page, shown in Figure 4-20, is the first page that users will see. From this page, all contents of the CRC pages can be accessed, and the user may also navigate back to the RL home page by using the arrow button at the bottom of the page. At the top is a frame containing the RL logo and three navigation buttons. This frame is always visible for convenient navigation within the CRC web pages.



Welcome to the Certification of Reusable Software Components World Wide Web Pages

In January of 1994, Rome Laboratory began a thirty-month, exploratory development project entitled "Certification of Reusable Software Components" (CRC). Under the CRC contract, a Certification Framework (CF) for software components was developed which is sensitive to varying domains, business strategies and asset types. A cost benefit plan, an operational concept, and a suite of certification tools were defined. An automated demonstration of the Cost Benefit Model was also developed. A data collection guide and procedures for a certification field trial were developed, an initial field trial was conducted, and the results were analyzed and reported.

The aforementioned documents and the automated demo are available via these web pages.

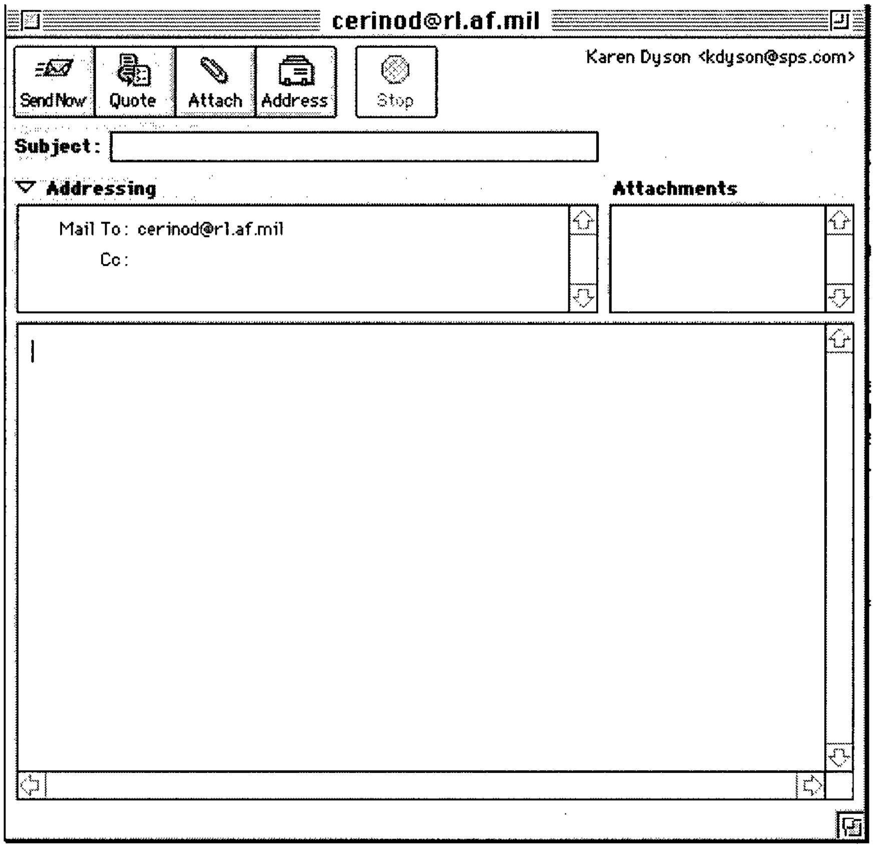
For more information about Rome Laboratory's research program, contact Deborah Cerino at cerinod@rl.af.mil.

- View the Executive Summary
- Download the CRC Documentation Suite
- Run the automated Cost Benefit Model Demo. You will need a java-enabled browser in order to run the demo.

Click here to go to the Rome Laboratory Home Page.

Figure 4-20. CRC Welcome Page

On the welcome page, an email link to the RL contact Deborah Cerino is provided. When the user clicks on this link, the browser automatically brings up a mail window such as the one shown in Figure 4-21. Users desiring more information can easily contact Ms. Cerino.



cerinod@rl.af.mil

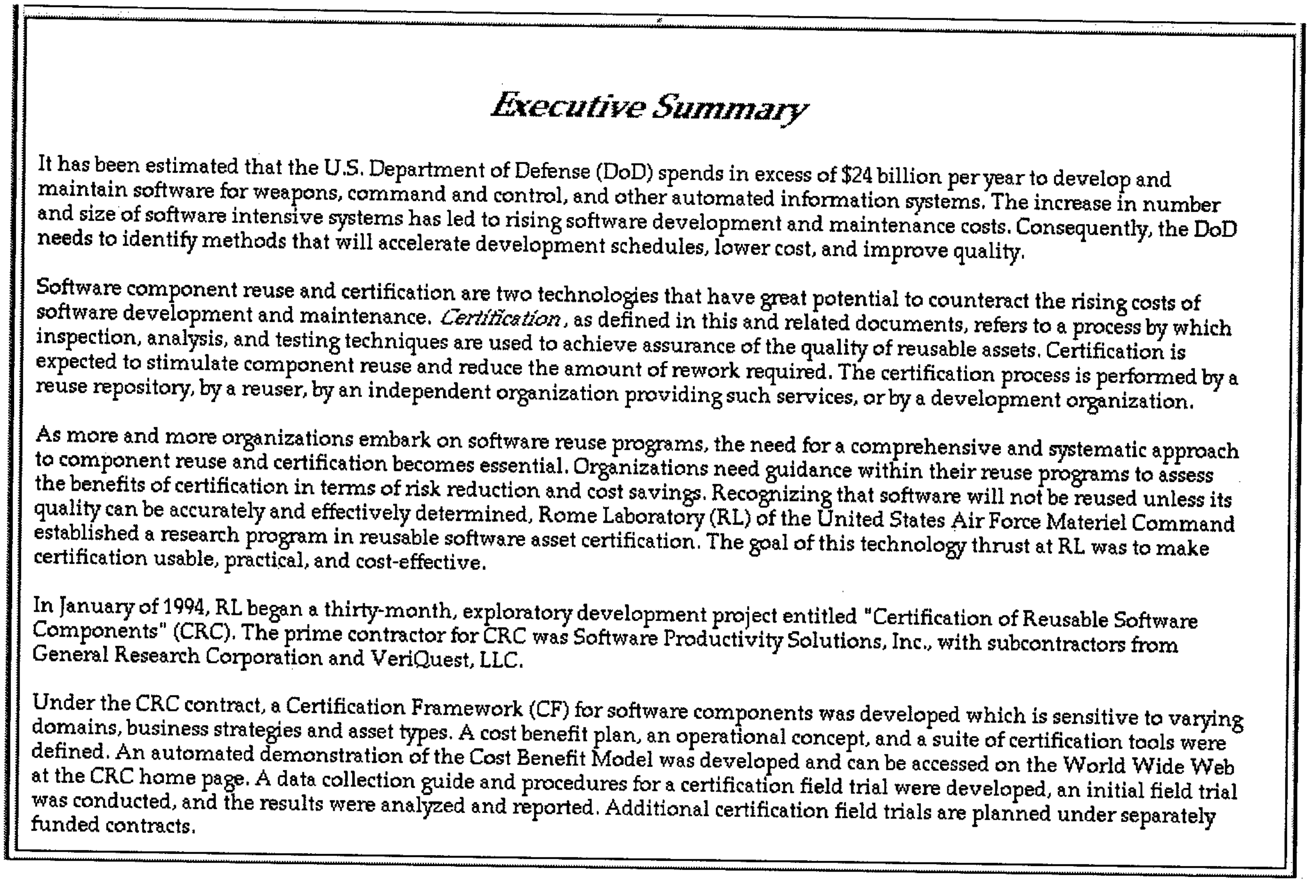
Karen Dyson <kdyson@sps.com>

Mail To: cerinod@rl.af.mil

Cc:

Figure 4-21. CRC Email Link

Upon selection of the Executive Summary link on the welcome page, the executive summary is displayed in the bottom frame of the browser window, as shown in Figure 4-22.



Executive Summary

It has been estimated that the U.S. Department of Defense (DoD) spends in excess of $24 billion per year to develop and maintain software for weapons, command and control, and other automated information systems. The increase in number and size of software intensive systems has led to rising software development and maintenance costs. Consequently, the DoD needs to identify methods that will accelerate development schedules, lower cost, and improve quality.

Software component reuse and certification are two technologies that have great potential to counteract the rising costs of software development and maintenance. *Certification*, as defined in this and related documents, refers to a process by which inspection, analysis, and testing techniques are used to achieve assurance of the quality of reusable assets. Certification is expected to stimulate component reuse and reduce the amount of rework required. The certification process is performed by a reuse repository, by a reuser, by an independent organization providing such services, or by a development organization.

As more and more organizations embark on software reuse programs, the need for a comprehensive and systematic approach to component reuse and certification becomes essential. Organizations need guidance within their reuse programs to assess the benefits of certification in terms of risk reduction and cost savings. Recognizing that software will not be reused unless its quality can be accurately and effectively determined, Rome Laboratory (RL) of the United States Air Force Materiel Command established a research program in reusable software asset certification. The goal of this technology thrust at RL was to make certification usable, practical, and cost-effective.

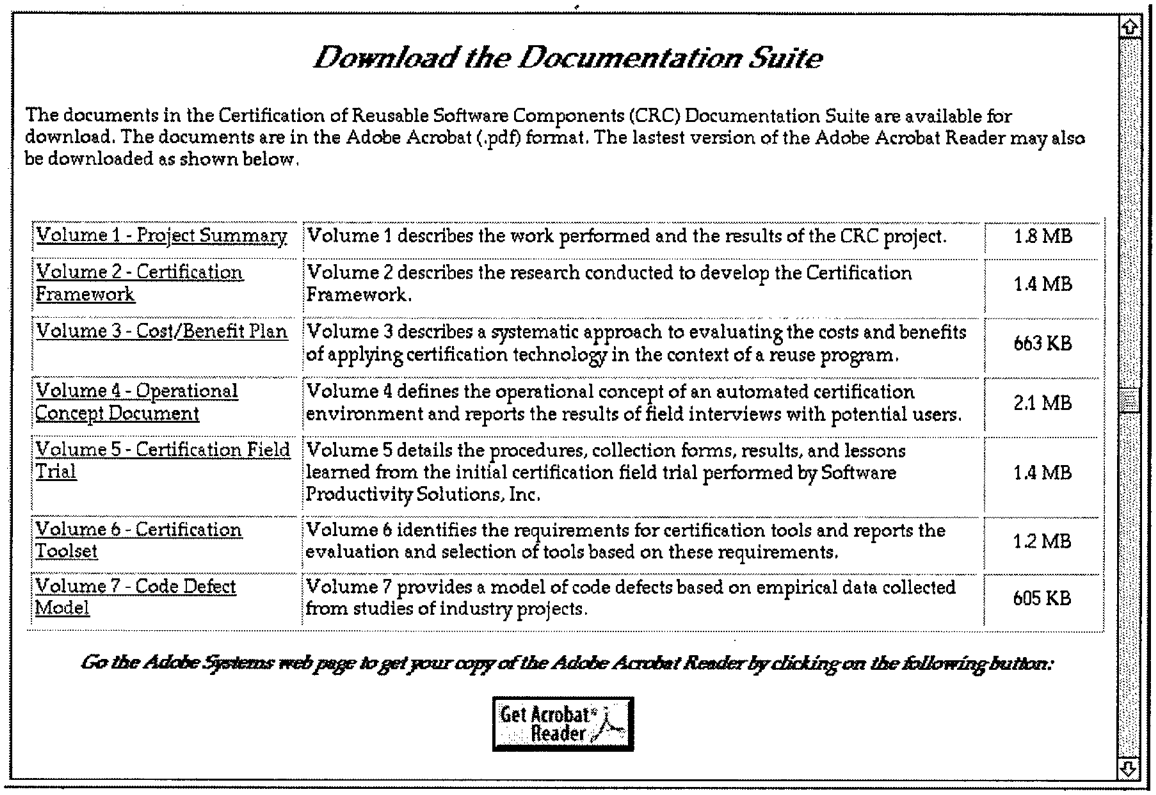
In January of 1994, RL began a thirty-month, exploratory development project entitled "Certification of Reusable Software Components" (CRC). The prime contractor for CRC was Software Productivity Solutions, Inc., with subcontractors from General Research Corporation and VeriQuest, LLC.

Under the CRC contract, a Certification Framework (CF) for software components was developed which is sensitive to varying domains, business strategies and asset types. A cost benefit plan, an operational concept, and a suite of certification tools were defined. An automated demonstration of the Cost Benefit Model was developed and can be accessed on the World Wide Web at the CRC home page. A data collection guide and procedures for a certification field trial were developed, an initial field trial was conducted, and the results were analyzed and reported. Additional certification field trials are planned under separately funded contracts.

Figure 4-22. CRC Executive Summary Page

The user can also download PDF versions of the seven-volume CRC document suite by clicking on the Download Documents link on the welcome page, or by clicking the Documents button. The PDF format files are browsable and printable from Adobe Acrobat Reader. The Acrobat Reader application is available at no charge from Adobe. There is a link to the Adobe web site at the bottom of this page for users who wish to download this application.

The seven downloadable documents are listed in a table (see Figure 4-23) along with a brief description of their contents and a file size. When the user clicks on the link to a file, the file is downloaded from the server to his local (client) computer.



Download the Documentation Suite

The documents in the Certification of Reusable Software Components (CRC) Documentation Suite are available for download. The documents are in the Adobe Acrobat (.pdf) format. The lastest version of the Adobe Acrobat Reader may also be downloaded as shown below.

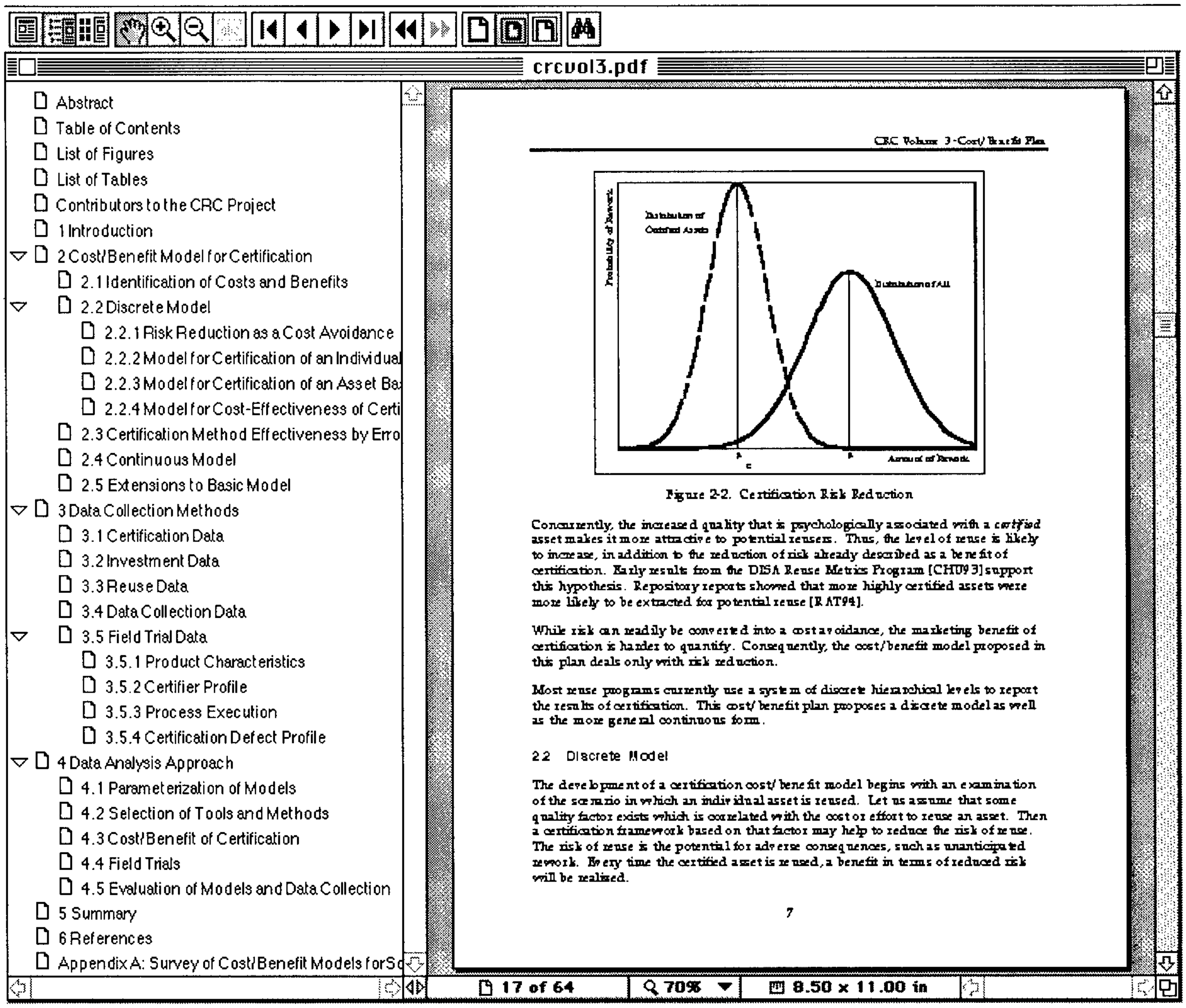
Volume 1 - Project Summary	Volume 1 describes the work performed and the results of the CRC project.	1.8 MB
Volume 2 - Certification Framework	Volume 2 describes the research conducted to develop the Certification Framework.	1.4 MB
Volume 3 - Cost/Benefit Plan	Volume 3 describes a systematic approach to evaluating the costs and benefits of applying certification technology in the context of a reuse program.	663 KB
Volume 4 - Operational Concept Document	Volume 4 defines the operational concept of an automated certification environment and reports the results of field interviews with potential users.	2.1 MB
Volume 5 - Certification Field Trial	Volume 5 details the procedures, collection forms, results, and lessons learned from the initial certification field trial performed by Software Productivity Solutions, Inc.	1.4 MB
Volume 6 - Certification Toolset	Volume 6 identifies the requirements for certification tools and reports the evaluation and selection of tools based on these requirements.	1.2 MB
Volume 7 - Code Defect Model	Volume 7 provides a model of code defects based on empirical data collected from studies of industry projects.	605 KB

Go the Adobe Systems web page to get your copy of the Adobe Acrobat Reader by clicking on the following button:

Figure 4-23. CRC Document Download Page

If the user has Acrobat Reader installed and established as a helper application in his browser, then Acrobat Reader will automatically be invoked upon completion of the download. As shown in Figure 4-24, from Acrobat Reader, the user can browse the entire document and can print a high-quality hardcopy on his own printer. Bookmarks have been established for easy navigation throughout the document. These are shown on the left-hand side of the Acrobat Reader window. At any time, the user may choose to exit Acrobat Reader or place that window in the background and return to the browser.

If the user has an Acrobat Reader plug-in configured to his browser, then Acrobat Reader would instead be invoked within the bottom frame of the browser window. If the user does not have Acrobat Reader installed, he may simply download the desired files and browse them at a later time.



CRC Volume 3 - Cost/Benefit Plan

Figure 2-2. Certification Risk Reduction

Concurrently, the increased quality that is psychologically associated with a *certified* asset makes it more attractive to potential reusers. Thus, the level of reuse is likely to increase, in addition to the reduction of risk already described as a benefit of certification. Early results from the DISA Reuse Metrics Program [CHU93] support this hypothesis. Repository reports showed that more highly certified assets were more likely to be extracted for potential reuse [RAT94].

While risk can readily be converted into a cost avoidance, the marketing benefit of certification is harder to quantify. Consequently, the cost/benefit model proposed in this plan deals only with risk reduction.

Most reuse programs currently use a system of discrete hierarchical levels to report the results of certification. This cost/benefit plan proposes a discrete model as well as the more general continuous form.

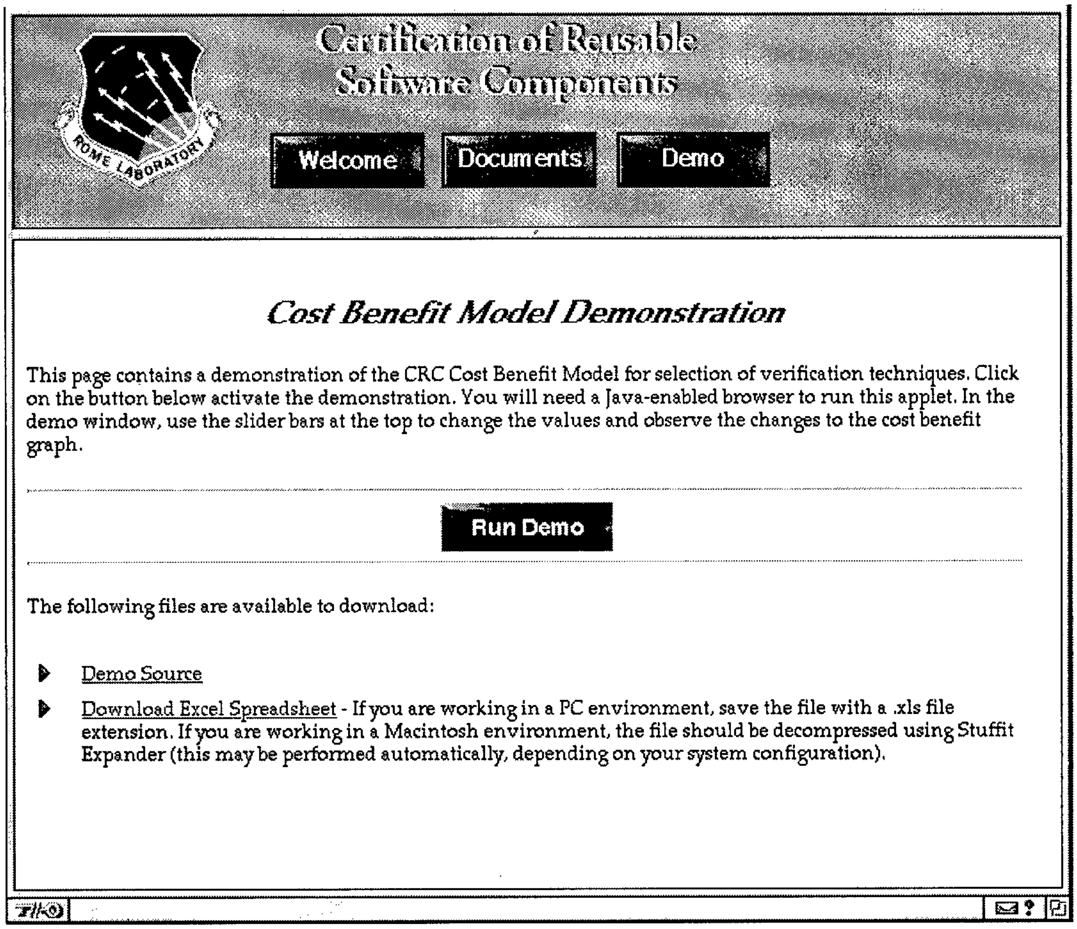
2.2 Discrete Model

The development of a certification cost/benefit model begins with an examination of the scenario in which an individual asset is reused. Let us assume that some quality factor exists which is correlated with the cost or effort to reuse an asset. Then a certification framework based on that factor may help to reduce the risk of reuse. The risk of reuse is the potential for adverse consequences, such as unanticipated rework. Every time the certified asset is reused, a benefit in terms of reduced risk will be realized.

7

Figure 4-24. Adobe Acrobat Reader Window Showing CRC Volume 3 Document

Back on the welcome page, there is a link to the Demonstration. By clicking this link, or the Demo button, the demonstration page will be displayed as shown in Figure 4-25. If the user does not have a Java-enabled browser, the Run Demo button will not be shown. In this case, the user may download an Excel spreadsheet version of the demonstration to run on his local computer. The user can also download a compressed file containing the Java source code.



Cost Benefit Model Demonstration

This page contains a demonstration of the CRC Cost Benefit Model for selection of verification techniques. Click on the button below activate the demonstration. You will need a Java-enabled browser to run this applet. In the demo window, use the slider bars at the top to change the values and observe the changes to the cost benefit graph.

The following files are available to download:

- Demo Source
- Download Excel Spreadsheet - If you are working in a PC environment, save the file with a .xls file extension. If you are working in a Macintosh environment, the file should be decompressed using Stuffit Expander (this may be performed automatically, depending on your system configuration).

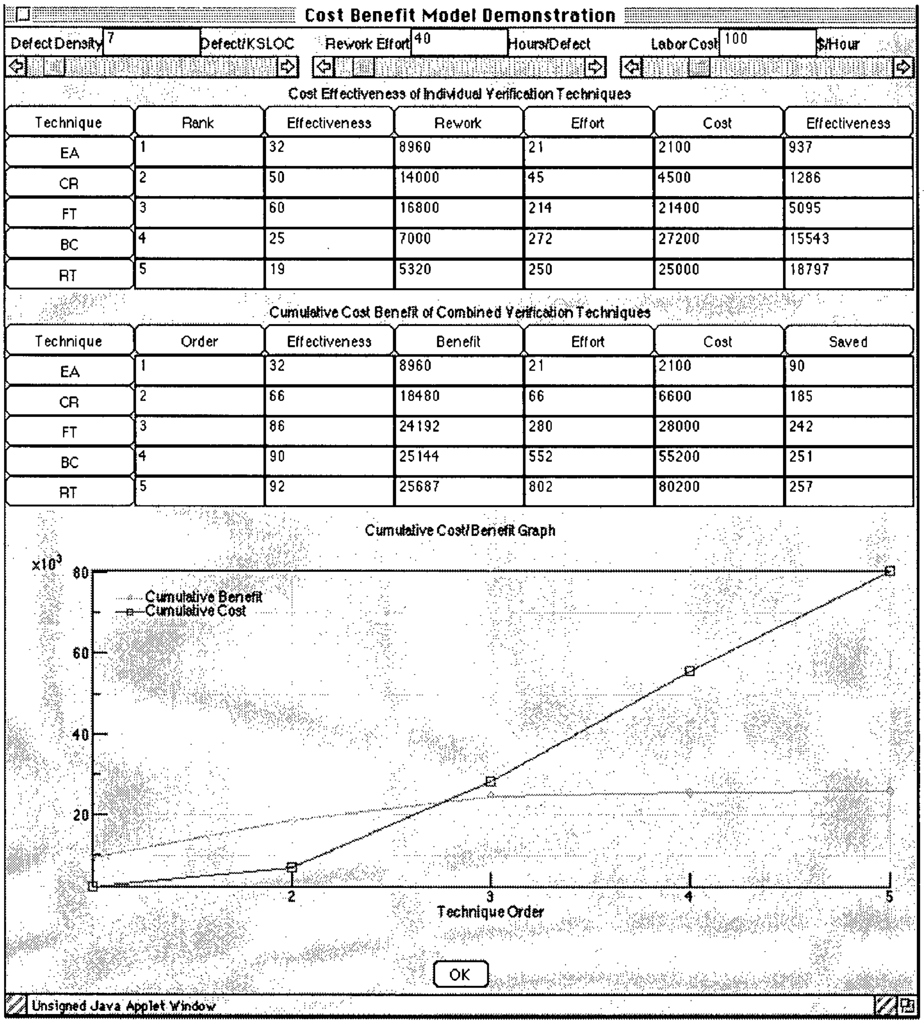
Figure 4-25. CRC Demonstration Page

When the user clicks the Run Demo button, the server will download the applet to the client computer and the browser will control its execution. The Java applet will start and a new window will open as shown in Figure 4-26. There are three input variables at the top of the window that the user may modify by moving the slider bars: defect density, rework effort per defect, and labor cost. As these values are changed, the values in the tables will be recalculated and the graph redrawn.

The purpose of this demonstration is to illustrate the CRC cost/benefit analysis of verification techniques based on the aggregate defect model discussed in more detail in section 4.3.1. Five techniques are shown: Error & Anomaly Analysis (EA), Code Review (CR), Functional Testing (FT), Branch Coverage Testing (BC), and Random Testing (RT). The cost of a technique is the effort to apply it; investment costs such as tool purchase cost and training effort are not considered in this analysis. The benefit of a technique is the cost of rework avoided based on the technique's effectiveness at detecting defects. Technique application cost and effectiveness data is taken from [MCC92].

Techniques are applied in order from the most cost-effective to the least cost-effective. Cost effectiveness is calculated as (average) cost per defect found per thousand lines of code. The cumulative combined effectiveness is shown in the lower table.

As the user modifies the input values of defect density or rework effort, the break even point of cost/benefit graph (where the two curves intersect) will move. The break even point represents the benefit equal to the cost incurred. Beyond this point, to the right, the cost exceeds the benefit. When finished, the user can exit the applet by closing the window or clicking the OK button at the bottom of the window. At this point he is returned to the CRC demonstration page in the browser.



Cost Benefit Model Demonstration

Defect Density 7 Defect/KSLOC Rework Effort 40 Hours/Defect Labor Cost 100 $/Hour

Cost Effectiveness of Individual Verification Techniques

Technique	Rank	Effectiveness	Rework	Effort	Cost	Effectiveness
EA	1	32	8960	21	2100	937
CR	2	50	14000	45	4500	1286
FT	3	60	16800	214	21400	5095
BC	4	25	7000	272	27200	15543
RT	5	19	5320	250	25000	18797

Cumulative Cost Benefit of Combined Verification Techniques

Technique	Order	Effectiveness	Benefit	Effort	Cost	Saved
EA	1	32	8960	21	2100	90
CR	2	66	18480	66	6600	185
FT	3	86	24192	280	28000	242
BC	4	90	25144	552	55200	251
RT	5	92	25687	802	80200	257

Figure 4-26. CRC Demonstration Applet Window.

4.3.1 Cost/Benefit Model for Aggregate Defects

This section describes the cost/benefit model for selection of certification (or verification) techniques based on an aggregate defect model. This is a revision to the original cost/benefit model described in the CRC Volume 3 Cost/Benefit Plan in section 2.3 of that document. This revised model differs from the original model in that it eliminates the defect profile that characterizes defects by types (i.e., Computational, Data, Interface, Logic and Other). Instead, this revised model considers just aggregate defects, or total defects of all types.

The original cost/benefit model for technique selection was based on the premise that by breaking the defects down into types we would be better able to select the most cost-effective techniques. This premise includes two major assumptions as follows:

- A profile of expected defect types can be constructed from historical data
- The effectiveness of certification techniques at detecting defects of the different types is known.

Research was conducted on the CRC effort to establish the best available data from published studies to support the above two assumptions, and this was documented in CRC Volume 5 Code Defect Model. The data was applied to an Excel spreadsheet version of the model to rank the techniques in order of cost-effectiveness when applied in sequence, as would be the case in a multi-step certification process.

The model used the following algorithm:

1. Rank the techniques in order of their individual cost-effectiveness, where cost-effectiveness is calculated as the average cost per defect found per 1000 lines of code.

2. Apply the most cost-effective technique (from step 1) first, calculating the cost to apply and the benefit in terms of rework avoided.

3. Re-compute the defect profile after having applied the previous technique, and re-rank the remaining techniques by cost-effectiveness.

4. Apply the most cost-effective of the remaining techniques, calculating the cumulative cost to apply and cumulative benefit. Cumulative cost is calculated as the sum of the cost to apply the individual techniques. Cumulative effectiveness is equal to the sum of the cumulative effectiveness of the prior step's techniques plus the effectiveness of this step's minus the product of these two values.

5. Repeat steps 3 and 4 until all techniques have been applied (i.e., ranked).

Using this algorithm on the available data resulted in a ranking of techniques that was the same as the McCall study of technique effectiveness [MCC92] which did not break defects down by type. Thus the defect type breakdown *did not add any value.* We postulate that is because the differences labor costs (effort to apply) far outweigh the differences in effectiveness among the techniques considered. For example, automated static analysis is by far the least labor-intensive, and testing techniques are the most labor-intensive.

One could imagine a case with an atypical defect profile, such as a profile that is heavily skewed toward one particular type of defect, where the above model would perhaps lead to a different ranking of techniques. But for a typical defect profile, the added complexity of the above model is not needed.

Thus we collapsed the defect profile by types into an aggregate defect density for the revised cost/benefit model which is illustrated in the automated demonstration. The same basic algorithm applies, except that it is no longer necessary to recompute the ranking of remaining techniques after having applied a technique, since the application of a technique no longer impacts the profile of defects remaining.

Research by George Stark of the MITRE corporation reported in [STA96] suggests that the CRC defect categorization (Computation, Data, Interface, Logic and Other) may not be the most effective breakdown for explaining the difference in rework cost per defect. When the rework cost per defect category varies by an order of magnitude or more, then the categorization scheme is an effective discriminator. For example, if defects of Type A require 10 times more effort to repair than other types, a cost/benefit analysis would clearly favor techniques targeted at Type A defects. Likewise, a process improvement initiative would be targeted to prevention of Type A defects.

5.0 Conclusions

This section presents our conclusions resulting from the Certification Framework Validation for Reusable Assets effort documented in this Final Technical Report. Section 5.1 contains a summary of the project. Section 5.2 contains the lessons learned as a result of this effort. The final section, Section 5.3, presents ideas for future research identified as a result of this effort.

5.1 Project Summary

This FTR provides a comprehensive, cumulative, and substantive summary of the progress and significant accomplishments achieved on the project titled "Certification Framework Validation for Reusable Assets." The efforts of the ATD project advanced reuse and certification technologies, as well as the state of the art in software certification framework validation. These advances are identified in Figure 5-1 and show how each was built upon the accomplishments of the previous CRC project.

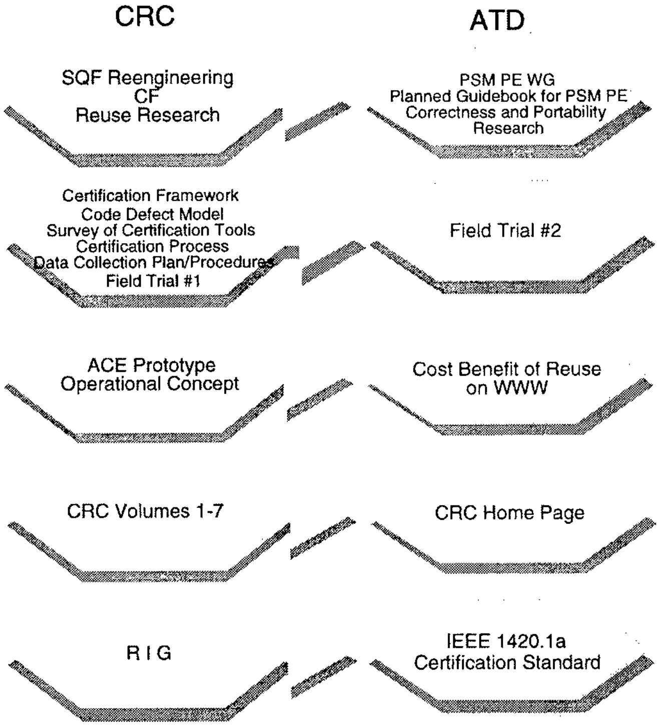


Figure 5-1. The Results of the ATD Project Built Upon the Accomplishments of CRC.

Even though we found that reuse and certification are still relatively immature in industry practice, the ATD project demonstrated tangible work products. The work products together with the lessons learned described in the subsection that follows provide a sound foundation for future research topics.

The two main objectives established for this effort were achieved as a result of the tasks performed. The two main objectives were as follows:

1. Upgrade and expand the application of the RL Software Quality Framework (SQF).
2. Further develop, apply and validate the RL Certification Framework (CF) initially developed under the CRC contract.

First Objective

To address the first objective, we identified needed SQF framework upgrades, and developed a re-engineering approach. The following aspects of the framework were redesigned:

- Framework structural hierarchy—splitting the measurement portion of the framework away from the guidance
- Streamlined the tailoring approach from factor selection to implementation of quality assessment
- Application guidance—recommending that the SQF inspections be merged into the existing software development process rather than a set of stand-alone activities

We also extracted the following useful techniques from the SQF applicable to reuse certification:

- Guidelines for building correctness into reusable assets
- Ada and C++ code inspection checklists
- Automated Ada style guideline checks
- Complexity measures for a defect prediction model

The code inspection checklists and automated Ada style guideline checks were validated by use in the certification field trials. The complexity measures were found to be effective at predicting different defect types by MITRE's model calibration task.

Second Objective

To address the second objective, we conducted research into commercial organizations and reuse libraries in parallel with the CRC contract in order to provide timely feedback to the development of the Certification Framework (CF) and the certification process. Our conclusion is that the CF technology is significantly ahead of the state-of-the-practice of organizations that are involved in reuse.

In analyzing the BLSM I problem reports we confirmed that defect data that is collected for Air Force MIS systems can be used to generate a defect profile for the CRC cost/benefit model.

We also performed a second certification field trial for a C++ asset. The first field trial was performed with an Ada asset, but C++ is more commonly used in commercial organizations than is Ada. An entirely new set of certification tools had to be selected for C++.

We consider that the second field trial was a success, as was the first field trial. After completion of both field trials, we determined the following findings:

- Certification process is repeatable.
- Certification process is understandable by other Certifiers.
- The effort required to perform the Certification process for C++ code components is similar to the effort required for Ada code components.
- Tools are available for C++ certification that provide roughly the same capabilities as for Ada.

We also created the CRC web pages including an interactive demonstration of the CRC cost/benefit model for verification technique selection. These pages enhance technology transfer by making the CRC technologies visible and accessible to both researchers and practitioners.

5.2 Lessons Learned

Many valuable lessons were learned in the course of this effort. This section presents the lessons learned in three categories: SQF, field trials & certification process, and pilot sites.

5.2.1 SQF-Related Lessons Learned

The guidelines approach used to re-engineer the SQF was found to provide significant added value by incorporating a "quality blueprint", or guidance on how to build quality into software, into the framework. This added guidance rounds out the information contained in the SQF and transforms it into a more complete quality approach. We believe that extending this work by developing guidance for all of the SQF quality factors would be a very worthwhile contribution to software product engineering.

In performing the research to develop guidelines for the quality factor Portability, we were surprised by the lack of a complete approaches and generalized guidance, even in textbooks titled "Portability". For many quality factors, there is little available in the way of guidance, and what is available is scattered throughout many sources rather than collated into a coherent whole.

Another omission is in the availability of guidance for writing requirements, especially quantitative specifications, for software quality factors. For example, how does one specify a level of maintainability? How much is required for a given situation? A similar need is the capability to estimate the feasibility of achieving such requirements. This capability is essential in performing the trade-offs between functional and quality requirements. For example, how much effort will be required to construct a system with a reliability of 10^{-9}? Cost estimation models that incorporate these quality requirements are needed.

5.2.2 Certification Process and Field Trial Lessons Learned

The following section summarizes the lessons learned from the certification process and field trial. Additional details are found in section 4.1.1 of this document.

With regard to the choice of the component language, we found that the C++ language is not standardized. Even though the software industry has two C++ standards, many other flavors exist. This presents challenges in application and interoperability of certification tools. These factors have a significant effect on the reusability of a C++ asset because they impact maintainability and portability.

With regard to the defect categories in the CRC defect model, we found them difficult to assign from the definition alone. The defect model and the field trial procedures could be improved by elaborating the definitions and providing examples.

With regard to the testing effort, we found that the design of the component under test greatly affects the effort required to test and the ability to successfully perform maximum branch coverage during structural testing.

With regard to the configuration of the certification environment, we found this activity was time-consuming and fraught with obstacles. The effort to install, learn, integrate, and apply tools to a particular component should not be underestimated. Vendor training and responsive technical support are requirements for high-end testing tools.

With regard to the code inspection checklists, we found that the checklists must be customized to the implementation language of the source code under inspection. There are many differences between languages, subtle and not so subtle, that are potential sources of errors. One main category of such differences is portability considerations, which tend to be very language-specific. Another category is coding style; each language seems to acquire its own style.

Code inspection checklists must also be customized to an individual project or development organization to enforce local practice or design decisions. For example, one project may declare that all file I/O must be done through a specific file I/O package to ensure consistency. In certification of reusable components, such customization is not an issue because an asset is generally certified in a non-project-specific context. It is important to recognize these customizations, however, when developing a generic checklist from example checklists.

Code inspection requires personnel knowledgeable in the implementation language as well as in the application area. Past studies of the effectiveness of the inspection technique have shown that an inspector's experience is the greatest variable in the effectiveness of the technique. This means that it is difficult to achieve uniform, consistent effectiveness with this technique. We conclude that by off-loading as much as possible of the inspection checklist into static analysis tools, the overall effectiveness would be increased.

Despite the many tools currently available, there are numerous potential target areas for static analysis still to be exploited. Often the very areas where automated tools excel are the tedious, and thus error prone, inspection items that are difficult for the human inspector to analyze in an effective and repeatable fashion. For example, there is no excuse for requiring a human inspector to deal with coding style checks. Checks for the use of non-portable or non-standard language features are also prime areas for automation.

5.2.3 Pilot Sites Lessons Learned

Our field trials have demonstrated that a certification process that does not include inspection and testing is not effective at finding defects. Yet this type of certification process requires experienced personnel, as well as a significant amount of time and effort. In consulting with the Air Force Reuse Center personnel at Maxwell AFB Gunter Annex, we found that this level of investment in certification is beyond the scope of what most reuse repositories are able to support because it just does not fit with their business model. Their approach, which emphasizes automated tools, is roughly equivalent to steps 1 and 2 (compilation and static analysis) of our certification process.

The commercial pilot site at Underwriters' Laboratory (UL) intends to act as a third-party certifier of devices that include a significant amount of software, or in which software is critical to safe operation. In their business model, their certification service would be funded by the product vendors. UL has reached the conclusion that UL itself would not be performing a certification process such as the process developed on the CRC contract. Instead, their role would be to ensure that the software developers are following appropriate standards and are using techniques that assure the required level of software quality.

From UL's perspective, two aspects of the certification research are of primary interest: the quantification of the effectiveness of verification techniques and the mapping of required effectiveness to specific types of software applications/domains. Thus we conclude that these are viable areas for further research.

5.3 Future Research

The lessons learned discussed above provide a direction for future research. For example, the following topics show promise as critical areas for additional research and merit additional funding:

- Develop cost estimation model including SQF quality factors
- Complete re-engineering of SQF for all quality factors
- Improve CRC defect model so that defect categories are better related to rework effort
- Develop additional static analysis tools for cost-effective code inspection, especially for Portability concerns
- Investigate tools and methods to improve code inspection effectiveness, such as by improving code comprehension
- Continue Certification Framework research focused on needs expressed by UL (i.e.,mapping CF to standards, and quantitative technique effectiveness information)
- Develop a Verification Technique Testbed consisting of well documented "bad code" benchmarks with known defects; this would be useful for tool evaluation and testing, research into development of new or improved techniques, and quantifying technique effectiveness

Since reuse and certification has proved to be a more challenging technology that originally thought by the software industry, we recommend that future research be couched in the context of product engineering and software process improvement. Product engineering and process improvement certainly includes reuse and certification, but also encompasses other critical technologies as well. We feel that the products generated from this modified direction can be applied to a wider audience in the context of product engineering and software process improvement. The DoD, as well as commercial companies, are in great need of advancements in these technologies to achieve a competitive advantage in today's demanding markets.

References

[BAL92] Baldwin, John T. "An Abbreviated C++ Code Inspection Checklist." October 27, 1992. http: //www.ics.hawaii.edu/~johnson/FTR/Bib/Baldwin92.html.

[BEA94] Beaulieu, M.F. and Fischer, L.P. "How to successfully identify defects during an inspection." *Proceedings of STC '94*, April 1994.

[BOE81] Boehm, Barry W., *Software Engineering Economics*, Prentice-Hall, Inc., 1981.

[BOE88] Boehm, Barry W., "A Spiral Model of Software Development and Enhancement", *IEEE Computer*, May 1988.

[BOW85] Bowen, T. P., et. al. "Specification of Software Quality Attributes". Technical Report RADC-TR-85-37, Rome Laboratory, February 1985.

[DST96] "C Code Review Checklist." http://dstc.qut.edu.au/~baker/www/sqg/C_Checklist.html.

[DUN84] Dunn, R.H. *Software Defect Removal.* McGraw-Hill, New York, 1984.

[DYS91A] Dyson, Karen A. "Quality Evaluation System (QUES)", Final Technical Report for Rome Laboratory, RL-TR-91-407, Volume II, NTIS AD-A2523-976, December 1991.

[DYS91B] Dyson, Karen A. "Quality Evaluation System (QUES)", Final Technical Report for Rome Laboratory, RL-TR-91-407, Volume I , NTIS AD-A2523-679, December 1991.

[EBE94] Ebenau, R.G. and Strauss, S.H. *Software Inspection Process.* McGraw-Hill, New York, 1994.

[FAG76] Fagan, M.E. "Design and code inspection to reduce errors in program development." *IBM Systems Journal*, Vol. 15, No. 3, 1976, pp. 182-211.

[FAG96] Fagan, Michael. "OLP Software Inspection." http://www-ols.fnal.gov:8000/ols/www/inspection.html#focus.

[FAU94] Faure, John. "Draft Standards for C++ Usage." Internal corporate software development standard for Software Productivity Solutions, Inc. September 8, 1994.

[GER95] Gerisch, Margaret. "Code Review Checklist." http://www.oswego.edu/~more/html/checklist2.html.

[GRC95] GRC International, Inc., *Using AdaQuest*, Version 2.2, July 1995FAG76]

[HAM80] Hamlet, R.G. and R.M. Haralick, "Transportable Package Software", *Software -- Practice and Experience* 10 (1980), pp. 1009-1027.

[HEN88] Henderson, J., *Software Portability*, Gower Technical Press, 1988.

[HEN90] Henderson-Sellers, Brian, and Julian M. Edwards, "The Object-Oriented Systems Life Cycle", *Communications of the ACM*, , Vol. 33, No. 9, September 1990.

[HUM95] Humphrey, Watts. *A Discipline For Software Engineering.* SEI Series in Software Engineering. 1995.

[IEE91] IEEE Standard Glossary of Software Engineering Terminology, IEEE Std. 610.12-1990, Feb. 1991.

[JLC96] Joint Logistics Commanders, Joint Group on Systems Engineering, *Practical Software Measurement: A Guide to Objective Program Insight.* Version 2.1, March 27, 1996.

[KAN95] Kan, Stephen H., *Metrics and Models in Software Engineering*, Addison-Wesley Publishing Company, 1995.

[KOE92] Koenig, Andrew. "Checklist for Class Authors." *The C++ Journal.* Volume 2. No. 1. 1992.

[KOE95] Koenig, Andrew. "Working Paper for Draft Proposed International Standard for Information System—Programming Language C++." Doc No. X3J16/95-0185 WG21/NO785. September 26, 1995.

[LEC86] Lecarme, O. and M.P. Gart, *Software Portability*, 2nd ed., McGraw-Hill, 1986.

[LIN95] Linthicum, David, "Portability Pitfalls: Include Increased Cost, Potential Dissatisfaction", *Application Development Trends*, May 1995.

[MAR83] Martin, James and Carma McClure, *Software Maintenance: The Problem and Its Solution*, Prentice-Hall, 1983.

[MCC77] McCall, Jim A., et. al., "Factors in Software Quality", Final Technical Report, RADC-TR-77-369, Rome Air Development Center, Griffiss AFB, New York, November 1977.

[MCC82] McCracken, Daniel D. and Michael A. Jackson, "Life Cycle Concept Considered Harmful", *ACM SIGSOFT Software Engineering Notes*, Vol. 7, No. 2, April 1982.

[MCC92] McCall, James A. et. al., "Software Reliability, Measurement and Testing", Software Reliability and Test Integration, Final Technical Report RL-TR-92-52, Vols. I & II, Rome Laboratory, April 1992.

[MCC96] McCabe, Thomas. "McCabe OO Tool." Presentation materials from On-Site Tutorial. 1996.

[MOO90] Mooney, J.D., "Strategies for Supporting Application Portability", *IEEE Computer* , Nov. 1990, pp. 59-70.

[MOO93] Mooney, J.D., "Issues in the Specification and Measurement of Software Portability", TR 93-6, URL: http://www.cs.wvu.edu/~jdm/research/portability/reports/TR_93-6_ToC.html.

[NAS94] Angellatta, R. et. al. "Software Verification Plan for GCS," NASA Langley Research Center, Hampton, VA, December 1994.

[ONE88] O'Neill, D. and Ingram, A.L. "Software Inspections Tutorial." In *SEI Technical Review*, Software Engineering Institute, Pittsburgh, 1988.

[POT94] Potts, Stephen and Timothy S. Monk. *Borland C++ By Example.* Que Corporation. ISBN: 1-56529-756-3. 1994.

[REI95] Reilly, John P. and Erran Carmel, "Does RAD Live Up to the Hype?, *IEEE Software*, September, 1995.

[ROY70] Royce, W.W., "Managing the Development of Large Software Systems: Concepts and Techniques", *Proceedings WESCON*, August 1970.

[SAI89] SAIC, CSI, Inc., and SPS, Inc., Quality Evaluation System (QUES) Quality Framework Review, Interim Technical Report ,Volumes I-III, for Rome Laboratory, F30602-88-C-0019, June 1989.

[SAX85] Saxena, S. and Field, J.A., "Portable Real-Time Software for 8-bit Microprocessors", *Software -- Practice and Experience* 15 (1985), pp. 227-303.

[SCH93] Scheper, Charlotte, et. al. "Certification of Reusable Software Components". Draft Technical Report Prepared for Rome Laboratory, F30602-92-C-1058, January 1993.

[SKA94] Skazinski, Joseph, "Porting Ada: A Report from the Field", *IEEE Computer*, Oct. 1994.

[SOF95] Software Productivity Solutions, Inc. "Task Area: Software Quality Framework." Interim Technical Report. Data & Analysis Center for Software, Subcontract No. P48124 under Prime Contract No. F30602-92-C-0158. October, 1995.

[SOF96] Software Productivity Solutions, Inc. "Certification of Reusable Software Components: Volume 5, Certification Field Trial." Contract No. F30602-94-C-0024. United States Air Force, Rome Laboratory. Rome, NY. June 24, 1996.

[SOM92] Sommerville, I., *Software Engineering*, 4th ed., Addison-Wesley, 1992.

[SPC95] Software Productivity Consortium, *Ada 95 Quality and Style: Guidelines for Professional Programmers*, Version 01.00.10, October 1995.

[SPS94F] Software Productivity Solutions (SPS), Inc., Reuse Based Software Quality Framework for Certification (RC-SQF), Final Technical Report for Rome Laboratory, Prime Contract F30602-92-C-0158, Subcontract P48124, December 1994.

[SQF95] Software Quality Framework as Implemented in QUES Release 1.5, Software Productivity Solutions, Inc., Indialantic, FL, February 1995.

[STA96] Stark, George. "Stark Reality", *Measure Up!* Newsletter, Software Productivity Solutions, Inc., Volume 2 Number 4, October 1996.

[THO96] Thomas, William, "Predictive Models for Categorizing Error-Prone Software Modules", Final Technical Report, MTR 96W0000071, The MITRE Corporation, McLean, VA, September, 1996.

[VAN95] Ger van Diepen. "General C++ Coding Standard at the NFRA." http://www.nfra.nl/~qvd/seg/CppStdDoc.html.

[WIC77] Wichmann, B. A., "Performance Considerations", *Software Portability*, Cambridge University Press, Cambridge, England, 1977.

Acronyms

ACE	Automated Certification Environment
AFORMS	Air Force Operation Resource Management System
AFRC	Air Force Reuse Center
ARPA	Advanced Research Projects Agency
ASCII	American Standard Code for Information Interchange
ASIS	Ada Semantic Interface Specification
ATD	Advanced Technology Demonsration
ATP	Approved to Proceed
BC	Branch Coverage Testing
BLSM	Base Level Systems Modernization
CASE	Computer-Aided Software
CCB	Configuration Control Board
CF	Certification Framework
COTS	Commercial-Off-the-Shelf
CP	Completeness
CR	Code Review
CRADA	Cooperative Research and Development Agreement
CRC	Certification of Reusable Software Components
CRU	Computer Resource Utilization
CS	Consistency
CSC	Computer Software Component
CSCI	Computer Software Configuration Item
CSU	Computer Software Unit
CUB	Common Utilities and Bindings
DARPA	Defense Advanced Research Project Agency
DCF	Data Collection Form
DID	Data Item Description
DISA	Defense Information Systems Agency

DOD	Department of Defense
EA	Error & Anomaly Analysis
FT	Functional Testing
FTR	Final Technical Report
GCSS-AF	Global Combat Support Systems-Air Force
GRCI	General Research Corporation International
HTML	Hypertext Markup Language
I/O	Input/Output
IDA	Institute for Defense Analysis
IEEE	Institute of Electrical and Electronics Engineers
ISO	International Standards Organization
IT	Information Technology
ITG	Independent Testing Group
IV&V	Independent Verification & Validation
KSLOC	Thousand Source Lines of Code (non-blank, non-comment)
LOGMOD-B	Logistics Module-Base Level
LUA	Library Unit Aggregation
MDS	Management Data System
MTTR	Mean Time to Repair
NA	Not Applicable
NASA/GSFC	National Aeronautics & Space Agency Goddard Space Flight Center
NSA	National Security Agency
NUWC	Naval Undersea Warfare Center
OO	Object-Oriented
OSD	Office of the Under Secretary of Defense
PDF	Portable Document Format
PM	Program Management
PSM	Practical Software Measurement
QA	Quality Assurance
QUES	Quality Evaluation System
R&D	Research and Development

RAPID	Reusable Ada Products for Information Systems Development
RAASP	Reusable Ada Avionics Software Packages
RAD	Rapid Application Development
RC-SQF	Reuse-Based Software Quality Framework
RIG	Reuse Library Interoperability Group
RL	Rome Laboratory
SBIS	Sustaining Base Information System
SEI	Software Engineering Institute
SEL	Software Engineering Laboratory
SLOC	Source Lines of Code
SPC	Software Productivity Consortium
SQF	Software Quality Framework
SQT 2	Software Quality Technology Transfer
STARS	Software Technology for Adaptable, Reliable Systems
UL	Underwriters' Laboratory
V&V	Verification & Validation
VPI	Virginia Polytechnic Institute
W.AVG	Weighted Average
WG	Working Group
WWW	Worldwide Web

Appendix A - PSM Working Group Meeting Minutes

Practical Software Measurement
Software Product Engineering

Summary of Working Group Meeting of 20-21 August 1996

This Workshop had the following objectives:

(1) Define the scope and boundaries of SPE.

(2) Identify SPE issues supported by or related to measurement.

(3) Recognize intended user base for SPE measurement products.

(4) Clarify relationships between PSM SPE effort and other DoD and Industry initiatives.

The following definition received consensus agreement as a working definition:

> Within a project SPE comprises those software engineering activities used to produce products that meet identified user needs.

Based upon the objectives, the participants discussed several topics and expressed consensus on the following key questions:

Q: Should the SPE Measurement effort be pursued?

A: Yes.

Q: Should vendors be included?

A: Not by invitation, but participation remains open.

Q: Do we have software product engineers in the group? Are more needed?

A: Yes, we have people experienced in engineering of software products but we could benefit by having more.

SPE, following the PSM Program Management measurement precedence, is based on the rationale of issues-driven measurement; i.e. the questions of what information is needed and why drive the decision as to how measurement is to be done.

Consider Figure 1 below. While measurement needed for program managers emphasizes Cost and Schedule, that needed for software product engineers should focus on the Functionality and Quality of the software products (as distinguished from the system products).

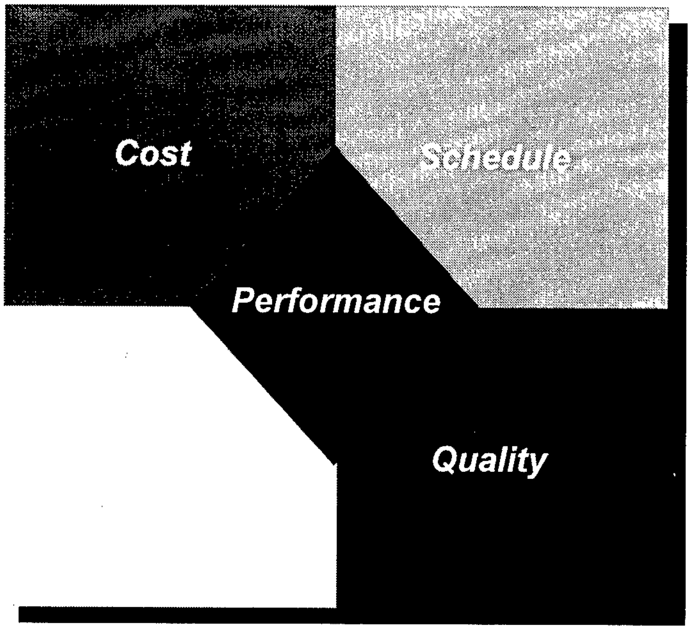


Figure 1

Estimation, prediction and assessment are all legitimate needs for measures in SPE. Very important is the specification of quality in the requirements to convey the concern much earlier than is typically the case now. (The TQM emphasis on customer satisfaction, which can be made only after considerable use, can bias the interest toward assessment only). Further, the scope of the requirements for SPE should include system implications, life cycle considerations and the scope of the "products."

The "customer base" for SPE can be categorized into three groups: Users (operational users, sponsor, functional user, independent testers), Developers (program manager, contractor's development manager, COTS selector, system integrator, designers, SQA group, programmers, development testers) and Maintainers (sustaining organization, in-service support agent).

Scope and boundaries of SPE were identified based on extensive discussion:

a. Project context in which SPE is done.
b. Exclude services, training, etc. as important but ancillary to SPE.
c. Systems engineering defines requirements and constraints on SPE.
d. Tradeoffs occur at system level and at the software level; while the former affect SPE the latter are fundamental to SPE.
e. Multiple users with different perspectives & informational needs.
f. Both activities (process) and products can be subjects for measure.

The IDEF-like Figure 2 illustrates the scope and boundaries emphasizing the iterative nature of the creative activities.

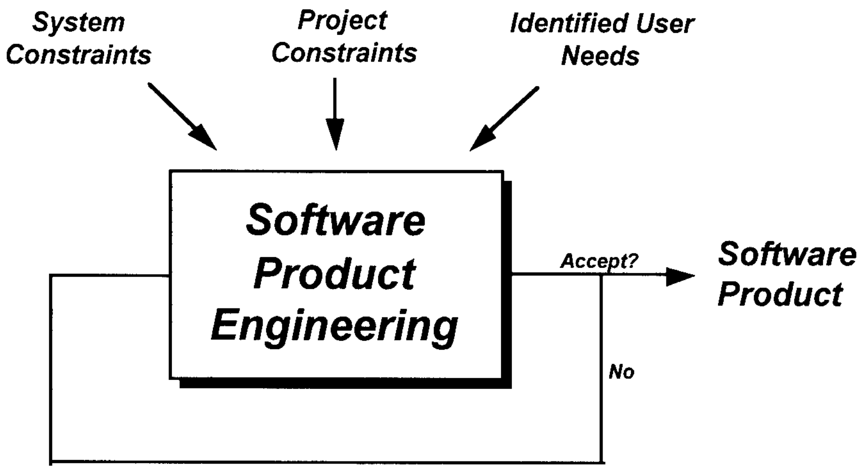


Figure 2

Functionality and quality are the initial attribute categories of interest for measurement in SPE. Clearly, decisions in these areas affect Cost and Schedule, the two primary categories for Program/Project Management. All four impact Performance. However, we contend for now that we see no issue involving the software product that forces considerations beyond functionality and quality.

The primary goal of the SPE Measurement effort is to develop and promote the understanding of the relationships among the four elements: Cost, Schedule, Functionality, and Quality. All contribute to the consequent determination of Performance.

Individuals, organizations, documents, etc. that could contribute: PSM Program/Project effort, Boehm (COCOMO II), SEI CMM, Personal CMM (Humphrey), Capers Jones' Enterprise Model, Best Practices Effort, Malcolm Baldridge Criteria, ISO Standards efforts, Function Points papers, Objectives/Principles/Attributes (Virginia Tech), Quality Factors (Rome Labs), NASA SEL work, GQM (Basili and Rombach), SMERFS (Farr), Design Metrics (Zage & Zage), Musa's Reliability Modeling, Gaffney's Defect Modeling, McCabe's Complexity work.

The next PSM-PE meeting was planned for October 30, 1996. The focus of the meeting will be the identification of specific product engineering issues and a related measurement framework.

Practical Software Measurement
Software Product Engineering

Summary of Working Group Meeting of 30 October 1996
Submitted on 6 November 1996

This Workshop had the following objectives:

(1) Review and discuss the results of the Workshop of 20-21 August 1996. These results focused on a preliminary technical approach to the overall effort and scoped and defined related issues.

(2) Identify the key software product engineering issues.

(3) Define a preliminary structure for categorizing the Software Product Engineering Measurement issues.

A issue was defined as a concern or a area where problems may occur. The definition of an issue should be consistent with PSM PM. The process for measurement established in the PSM PM will be adopted for PSM SPE. For example, Cost and Schedule can be considered as constraints upon Functionality and Quality. SPE shares similar concerns with Program Management (i.e., recognize problems early, identify and track risks, and satisfy constraints).

To help focus our discussion of issues, we revisited the concept diagram of PSM as represented in Figure 1. For SPE, we are excluding items address by PSM PM in the concept diagram.

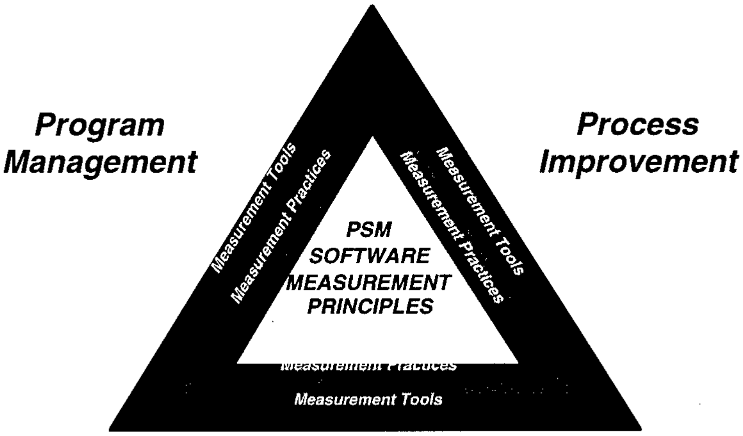


Figure 1. The Concept of Practical Software Measurement

By using a nominal group technique (i.e., structured brainstorming), the participants derived a list of issues and their meanings discussed. We used the following codes:

- F = Functionality
- Q = Quality
- A = Analysis/constraint
- P = Process
- B = Both Functionality and Quality
- T = Technical adequacy

Issues within a category or group should apply to all domains and methodologies. The groupings should be reconcilable with PSM for the PM. It was decided to raise the granularity of the grouping exercise and indicate if each issue is "**in**" = In the box of SPE or "**out**" = Outside of the box of SPE. The results of all these exercises are listed below.

- Defect recognition - **in** (Q) (i.e., when inserted, found, fixed)
- Feedback- **out** (P) (i.e., the necessity of feedback, during the development effort, to initiate process changes, to describe quality now, to predict quality downstream)
- Definition of quality - **in** (Q) (i.e., What do we mean by quality?)
- Completeness - **in** (F) (i.e., in terms of functionality, is all expected functionality present?)
- Cost effectiveness - **out** (A) (i.e., economical conformance to requirements [Deming])
- Initial cost estimate - **out** (A) (i.e., cost of product)
- Safety - **in** (B) (i.e., How safe is system, How safe is the software?)
- Ease of use - **in** (Q) (i.e., How easy is it to use?)
- Schedule - **out** (A) (i.e., planning and direction of schedule PSM PM)
- Requirements Understandability - **in** (F) (i.e., Do two people build the same software from the same requirements?)
- Real Time Performance - **in** (A), (F) (i.e., how the software works at run-time)
- Supportability- **in** (B) (i.e., after development, post-build, logistics, How easy is it to fix? Enhance? Patch?)
- Usability - **in** (i.e., ease of use)
- Quality Flow Through - **in** (i.e., components from one system to another, from product-to-product, from phase-to-phase, from requirements-to-documentation-to- system, What is the weakest link in the chain?)
- Reliability - **in** (i.e., defects in the release, defects acceptable for release, errors remaining)
- Maintainability - **in**
- Product Correctness - **in**
- Conflicting requirements - **in** (i.e., customer diversity or bias, potentially conflicting customer requirements)
- Hard versus soft or fuzzy functional requirements - **in** (i.e., some requirements are hard and fast, others are negotiable)
- Testability - **in**

- Definition of a failure - **in** (i.e., What is a failure? A failure to the implementor may be different for a failure to the user.)
- Readiness to deliver - **in** (i.e., product maturity, Is the product ready to deliver?)
- Reusability - **in** (i.e., How reusable is the software? How reusable is what we are developing?)
- Quality of development environment - **out** potential new issue (i.e., Are there bugs in the tools of the development environment?)
- Traceability - **in**
- Time to market (customer need) - **out** (A) (i.e., driven by customer need)
- Liability - **out**
- Are the evolved requirements really what the user intended?- **in** (i.e., requirements being what the user intends)
- Portability - **in**
- Affordability - **out**
- Efficiency of product - **in**
- Statusing - **out** (P) (i.e., measurement)
- Scalability - **in** (.i.e., expandability, Can you make it bigger to launch 25 missiles instead of 5?)
- Security - **in**
- Planning and estimating - **out**
- Precision - computational - **in**
- Complexity - **in**
- Integration of measurement data - **out** (P) (i.e., attributes, data)
- Degree of software reuse - **in** (i.e., incorporating reusable software may lead to duplicative structure underneath, issues may be different)
- Degree of COTS - **in**
- Re-scoping of requirements during development - **in** (i.e., requirements volatility)
- Conformance to development process - **out**
- Standard product quality ranges - **in** (A) (i.e., for different application areas, bounds, tolerances)
- Concurrent engineering - **unresolved** (P) (T) (i.e., can also apply to product, integrated process and product design)
- Predictability of attainment of quality objectives - **in** (i.e., when? How early?)
- How quickly can changes be field? - **out** (i.e., ability to make changes, Is it a measure?)
- MTTR - **in**
- Fault tolerance - **in** (i.e., redundancy)
- Personnel skills and experience - **out**
- Conformance to standards - **in**
- Rework - **in**
- Product Reviews - **out** (i.e., scheduled)
- Product Size - **in**
- Technical Adequacy - **n/a** (already an issue)
- Maintenance versus Development trade-offs - **out** (P) (i.e., when to do an activity?)
- Adequacy of documentation - **in**

- Stability of requirements - **in**
- Development Stability - **out** (i.e., of a working group, language, environment, tools, organization)
- Clarity - **in**
- Consistency - **in**
- Measurability - **out**
- Mission criticality - **out**
- Definition of process - **out**
- Effectiveness of IPTs - **out** (i.e., other development approaches)
- Development of Maintenance Methodology - **out**
- Prediction of defects - **in**
- User satisfaction - **in**
- Definition of quality requirements - **in**
- Maintaining and improving the process - **out**
- Technology injection and related risk - **out** (T)
- Capability of organization to deliver required quality and functionality - **out** (P) (i.e., match of organizational capabilities to requirements, production function)
- Economies of scale - **out**
- Interfaces - **in** (i.e., of software to people, integration)
- Measurement robustness - **out**
- Expandability - **delete** (i.e., same as scalability)
- O-O concerns - **out** (P) (i.e., how to measure)
- Acceptance criteria - **in** (i.e., fitness for use)
- Consistency across multiple organizations- **out** (P) (subcontractors, prime vs. subcontractors, users)
- Quality of COTS/NDI - **in**
- Subcontracting/outsourcing issues - **out** (P)
- Interoperability - **in**
- Common operating environment - **in**
- Configuration management - **out** (P)
- Change control - **out** (P)

It was recognized that some items on the list are at different levels.

A SPE framework has the following purposes:

- Recognizes product engineering issues
- Provides a tailoring mechanism to perform selection of issue-driven measurements
- Provides an organization for information and aids in understanding
- Is a tool or model for planning or management

- Contains issues and measures and a method for helping to select measures
- Shows mapping between issue and measure
- Provides a justification for including a measure because it addresses a particular issue
- Indicates the relationship of issues (i.e., If you are worried about a specific issue, then you may been to worry about certain related issues.)

A framework needs to be simple and understandable so that it is used; the framework must be practical and useful. The framework must employ common sense (i.e., rather than theoretical) and guide into the best action. The strategy for developing the framework should be top-down, since a bottom-up development would rely on immature measures.

Criteria or characteristics of a framework were developed:

- Applicable to different domains (i.e., AIS, weapons, C3I, other)
- Applicable to different development and design methodologies (i.e., Object-Oriented, others)
- Compatible with the PSM PM (Note: We may need to revisit PSM PM to reconcile differences)
- Begins with Functionality (size, growth and stability) and Quality (product quality), may discover additional group
- Focuses on measurable entities
- Allows for different sources of code (i.e., new, reused, COTS, etc.)
- Addresses different user needs (i.e., PMs, software engineers, etc., to be addressed in the use of measures, process related issues)

Our going forward action was to put issues into specific categories to create a strawman framework. This activity will be done off-line. It was recommended that the issues be separated into two groups; Functionality and Quality. Then those two groups should be further subdivided into categories.

The next PSM SPE WG was scheduled for 18 December 1996.

Practical Software Measurement
Software Product Engineering

Summary of Working Group Meeting of 18 December 1996
Submitted on 3 January 1997

This Workshop had the following objectives:

(1) Categorize the Product Engineering issues identified during the meeting of 30 October 1996

(2) Discuss the business strategy for the PE project

(3) Begin initial discussions of the PE project plan

Definitions of Quality and Functionality were reviewed. Functionality can be defined as the ability of the software system to meet the needs of the operational uses as specified in the system requirements. Quality is determined by two contributing factors: 1) the degree to which the product functionality meets the needs expressed in system requirements, and 2) how well that functionality meets the user's expectations at delivery time.

The following comments summarize the round table discussion of Functionality and Quality and their relationship to each other.

- It was suggested that Functionality can be practically defined as "What does the system do?" (i.e., the system does this…). Then, quality can be defined as "How well the system does this…?" Such a simplification was key to the WG's discussions since PSM PM's value has been to treat a complex subject and simplify it for users.
- A traditional definition of Quality is conformance to requirements, or economical performance to requirements. Quality can be seen as the degree to which the requirements have been realized.
- There was concern that definition of Functionality may actually be the definition of Quality. Adding new Functionality may improve the Quality of the product. Two illustrative examples were described (i.e., Microsoft's Windows 3.1 and Windows 95; two systems that provide determination of the pitch, roll, and yaw of a vehicle in two different ways to the user).
- It is important to determine the relationship between Quality and Functionality because their definitions may establish their relationships. If Functionality is defined as how well requirements are met, then Functionality affects the Quality of the product.
- Performance may be satisfied by Functionality, and Performance may define Quality. For example, in the submarine domain, Functionality is defined by Performance. Performance is considered a functionality requirement.
- Basing the definition of Functionality on requirements may present a dilemma since requirements may be dependent upon the current state of technology. Changing technology may cause the requirements to be a moving target.

- Functionality is relevant to testing (i.e., either the product passes the contract requirement, or it fails). The WG felt is appropriate to address the requirements that are documented, rather than implied. The WG elected to table the discussion concerning implied or perceived requirements.
- The expectations of Functionality may not meet the desired Quality.
- One may have 100% conformance to requirements, but may not have a quality product.
- Functionality and Quality can be viewed in light of the evolution of software and the "sell-off" of a system. An ECP may be initiated with respect to the current requirements which cannot address the customer's changed view. Quality measurement may support the decision to accept the product.
- Several researchers have documented work in software quality factors. For example, McCall, Boeing, and Rome Laboratory, have contributed to this body of work. These approaches use 'ilities which are difficult to map to quantifiable measures and attributes. Instead, it was suggested that Tom Gill's work may be a more appropriate approach for practical measurement. Gill asks the questions, "Is it testable? Is it measurable?" If not, eliminate.
- It was proposed that Functionality and Quality can be viewed as a Venn diagram as shown in Figure 1. The stated requirements are the intended Functionality, whereas the product is the realized Functionality. Quality can be considered the actual degree to which the product meets its requirements. This diagram implies that measures may be needed for both product functionality and product expectations. User needs may change resulting in a shift in the circles and their intersections.

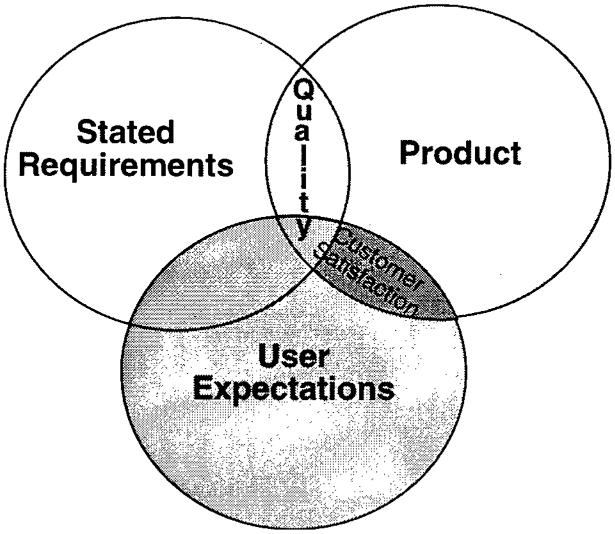


Figure 1. Products, Quality and Functionality

- The WG decided that, prior to determining PSM PE definitions of Functionality and Quality, work in the commerce arena and standards initiatives should be considered. Some definitions of terms are still moving targets, whereas others are baselined.
- If WG members have other definitions for Functionality and Quality that they would like the group to consider, Sean Arthur requested that they be sent to him and he will collate for distribution. He asked respondents to limit the definitions to five sentences.

In collating the individuals results of the categorization exercise, the following issues did not have a clear consensus concerning as to the categorization of a Functionality or Quality issue:

- efficiency of product
- product size
- acceptance criteria
- portability
- product correctness
- definition of a failure
- readiness to deliver
- complexity
- requirements volatility
- MTTR
- rework - restore may be better than repair
- stability of requirements
- consistency
- user satisfaction
- ease of use
- usability

Nonetheless, the WG felt that the categorization exercise was successful in that it verified that there are, indeed, two categories of issues, and they are Functionality and Quality. The WG also wanted to verify that the list of issues is complete, none are omitted, and issues do not pose redundancy. The WG still needs to verify that Performance's relationship with Cost, Schedule, Functionality, and Quality.

The definitions of Functionality and Quality may be dependent upon the user who is asking the question (i.e., user defined). It was pointed out that each user may define different concerns for Functionality and Quality. Each user may have different Common Quality Issues (CQIs), and the user may need to tradeoff Functionality and Quality. Of the three types of users previously defined (i.e., Users, Developers, Maintainers), the operational user is the focus of Product Engineering. Figure 2 illustrates these concepts.

It was proposed that software products can be divided into direct products and indirect products. Direct products are those items which must be created before the source code can be delivered. Indirect products are those items which support the development of the software, but are not necessary for its development. Examples of direct products include system requirements, software requirements, designs, and source code. Examples of indirect products include software development plans, configuration management plans, test plans, unit development folders, user's manuals, and installation manuals. The WG proposed to treat direct and indirect products differently from a measurement perspective.

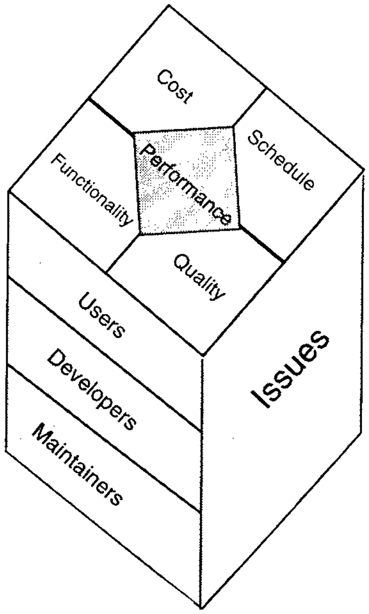


Figure 2. A user viewpoint of measurement issues

Internal product attributes are those characteristics of a product which can be measured solely in terms of the product itself. External product attributes are those characteristics of a product which can be measured only with respect to the product as it interacts with its environment. The attribute of size has different meanings for the entities of resources, processes, and products.

The following questions were raised:

Question: Should PSM PE address the product's process and resource issues, or just the products themselves?

Most process and resource attributes for products are already discussed in PSM PM.

Question: Should process and resource measures for products be discussed again in PSM PE?

Advantage: "One stop shopping"

Disadvantages: Redundancy, potential inconsistencies, must have both books

The following items were proposed as outside of the scope of Product Engineering:

- Resources (e.g., common operating environment, etc.)
- Processes (e.g., concurrent engineering, definition of quality, quality flow through, etc.)
- Products (e.g., clarity, consistency, etc.)

It was proposed that if an issue cannot be measured solely within the context of an product and its environment, it should be excluded from PSM PE. Process and resource issues should be dealt with only in PSM PM. Pointers between PE and PM may help increase circulation of both guidebooks. The WG agreed that top-down searches for measures to address perceived issues may be better than a bottom-up development of issues based upon the internal or external attributes. Planning may occur top-down whereas design may employ bottom-up techniques.

A plan for PSM PE and PM was presented as illustrated in Figure 3.

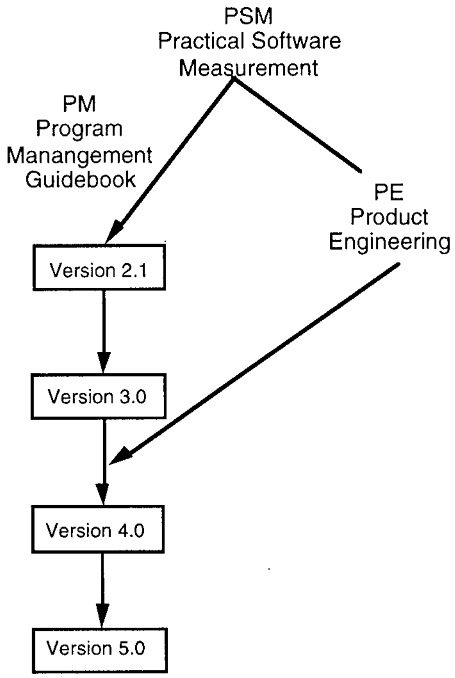


Figure 3. Plan for PSM PE as it relates to PSM PM

PSM's PM and PE can be applied to a single project, as well as across multiple projects. Chief Executive Officers (CEOs) are interested in addressing quality issues across projects to assess improvements within their organizations over time. Measuring for Process Improvement is an emerging technology. This technology supports a CEO's organizational view needs a practical approach since, to date, it has usually been addressed on a theoretical basis. Organizational Performance Measurement is anticipated to be a large market, interested participants could fund the multi-project view.

It was decided that the next step was to work as a subcommittee and the PE WG, as a whole, will act a reviewer of the PE subcommittee work products.

Appendix B - Code Inspection Checklist Sources

List of Sources for the Ada Checklist in this Appendix:

[BEA94] Beaulieu, M.F. and Fischer, L.P. How to successfully identify defects during an inspection. *Proceedings of STC '94,* April 1994.

[DUN84] Dunn, R.H. *Software Defect Removal.* McGraw-Hill, New York, 1984.

[EBE94] Ebenau, R.G. and Strauss, S.H. *Software Inspection Process.* McGraw-Hill, New York, 1994.

[FAG76] Fagan, M.E. Design and code inspection to reduce errors in program development. *IBM Systems Journal,* Vol. 15, No. 3, 1976, pp. 182-211.

[NASA94] Angellatta, R. et. al. Software Verification Plan for GCS, NASA Langley Research Center, Hampton, VA, December 1994.

[ONE88] O'Neill, D. and Ingram, A.L. Software Inspections Tutorial. In *SEI Technical Review,* Software Engineering Institute, Pittsburgh, 1988.

[SQF95] *Software Quality Framework as Implemented in QUES Release 1.5,* Software Productivity Solutions, Inc., Indialantic, FL, February 1995.

Source checklists follow in the above order.

[BEA94]	Beaulieu, M.F. and Fischer, L.P. How to successfully identify defects during an inspection. *Proceedings of STC '94*, April 1994.		
#	**Checklist Question**	**Used**	**Why Rejected**
1	Is the module preamble correct and complete?	✓	
2	Are all data declarations commented?	✓	
3	Are all data names descriptive enough?	✓	
4	Are variables initialized?	✓	
5	Are constant literals correct?	✓	
6	Are case statements explicitly defined?		Too vague: what does "defined" mean?
7	Are exception cases handled?	✓	
8	Are loop limits correct?	✓	
9	Are all branch conditions correct?	✓	
10	Are procedures called correctly?		Ada compiler will check
11	Are procedural return values handled?		Too vague: handled how?
12	Are parameter type declarations correct? consistent?		Ada compiler will check
13	Are variable type declarations correct? consistent?	✓	
14	Is the code an accurate representation of the design?		Design not available
15	Have all development standards been followed?		Too vague: standards not specified
16	Have all coding standards been followed?		Too vague: standards not specified
17	Is it readable?		Too subjective
18	Is it reusable?		Too subjective
19	Is it portable?		Portability is not a concern
20	Have timing, sizing, and throughput been addressed?		Efficiency is not a concern

[DUN84] Dunn, R.H. *Software Defect Removal.* McGraw-Hill, New York, 1984.		
Checklist Question	**Used**	**Why Rejected**
Is the compilation (or assembly) listing free of fault messages?		Not part of code inspection process
Have the deficiencies noted in the standards audit been corrected?		Not part of code inspection process
Do data definitions exploit typing capabilities to advantage?		Too subjective
Are mixed-mode expressions in violation of project standards?		Standards not specified: too vague
Do all pointers and indexes adhere to binding conventions?		Conventions not specified: too vague
Except for appropriate delimiters (for example, those in *for N in 1..10*), are constants expressed as parameters?		Not appropriate for Ada
For assembly language programs, have registers been saved on entry and restored on exit? Have stacks been properly initialized?		Not appropriate for Ada
For assembly language programs, have project standards for register use been observed?		Not appropriate for Ada
For branch points that correspond to the detailed design documentation, are the conditions sufficient? (that is, if n conditions are required for a decision, are all n present and properly accounted for?)		Design documentation not available
Are all conditions expressed in the correct sense (for example $A > B$ versus $B > A$)?	✓	
Do control constructs conform to structured programming standards?		Standards not specified: too vague
Are standards that restrict the depth to which loops and branches may be nested adhered to? Are loops and branches properly nested?		Standards not specified: too vague
Are indexes or subscripts properly initialized?	✓	
Are loop termination conditions invariably achievable?	✓	
Are any branch conditions mutually exclusive (such that they can lead to unreachable code)?	✓	
For assembly language programs, are the expected contents of memory used in lieu of their addresses?		Not appropriate for Ada
Do processes occur in the correct sequence?	✓	
Where applicable, are divisors tested for zero or noise?	✓	
Where applicable, are indexes, pointers, and subscripts tested against array, record, or file bounds?		Ada compiler will check

[DUN84] Dunn, R.H. *Software Defect Removal.* McGraw-Hill, New York, 1984.		
Checklist Question	**Used**	**Why Rejected**
Are imported data tested for validity? Are they ever reassigned except when they were transmitted for the express purpose of updating?	✓	
Do actual and formal interface parameter lists match?		Ada compiler will check
Do data declarations observe data boundaries implied by machine architecture, language, or user-defined declarations?		Not appropriate for Ada
Are all variables used? Are all output variables assigned?	✓	
Can any statements that are enclosed within loops be placed outside the loops without computational effect?	✓	
Is a more efficient mechanism for an input or output operation possible?		Efficiency is not of concern
Are the correct data being operated on at each statement?		Too vague
Are any labels unreferenced (a warning that something else may be amiss)?		Not appropriate for Ada
Are equations properly formed (e.g., if both *A* and *B* are to be divided, $(A+B)/C$ and not $A+B/C$)?	✓	
Can a connection to an external device result in an interminable wait?		Efficiency is not of concern
If there are requirements on execution time, will they be met?		Efficiency is not of concern
Does the code fit within its allocated storage? Local data?		Efficiency is not of concern
Have all the elements of the design been implemented as they were specified?		No design spec available
Has each function of the external specification for this code been correctly complied with?		No extenal spec functions available

[EBE94] Ebenau, R.G. and Strauss, S.H. *Software Inspection Process.* McGraw-Hill, New York, 1994.			
#	**Checklist Question**	**Used**	**Why Rejected**
	Data reference defects		
1	Is there a referenced variable whose value is unset or uninitialized?	✓	
2	For all array references, is each subscript value within the defined bounds of the corresponding dimensions?		Ada compiler will check
3	For all array references, does each subscript have an integer value? This may not be a defect, but it is a dangerous practice.	✓	
4	For all references through pointer variables, is the referenced storage currently allocated?	✓	
5	Does a storage area have alias names with different pointer variables? This is not a defect, but it is a dangerous practice.	✓	
6	Does the value of a variable have a type or attribute other than that expected? This is a common problem for storage referenced through pointers.		Ada compiler will check
7	Are there any explicit or implicit addressing problems? Examples are (a) the physical storage is smaller than the storage addressed in the program, and (b) the address is defined as byte address but used as bit address.		Not appropriate for Ada
8	Does the index of a string exceed its boundary?		Ada compiler will check
9	Are there any off-by-one defects in indexing operations or in subscript references to arrays?	✓	
	Data declaration defects		
1	Have all variables been explicitly declared? If a variable is not declared, is it understood that the variable is a global variable?		Not appropriate for Ada
2	If a variable is initialized in the declarative statement, is it properly initialized?	✓	
3	Is each variable assigned the correct length, type, and storage class?	✓	
4	Are there any variables that have similar names (e.g. *VOLT* and *VOLTS*)? This is not a defect, but it is a sign of confusion.	✓	
	Computation defects		
1	Are there any computations using variables having inconsistent data types (e.g., a Boolean variable in an arithmetic expression)?	✓	
2	Are there any mixed-mode (such as integer and floating-point) computations?	✓	

[EBE94] Ebenau, R.G. and Strauss, S.H. *Software Inspection Process.* McGraw-Hill, New York, 1994.			
#	**Checklist Question**	**Used**	**Why Rejected**
3	Are there any computations using variables having the same type but different length?	✓	
4	Is an overflow or underflow exception possible during the computation of an expression?	✓	
5	Is it possible for the denominator in a division operation to be zero?	✓	
6	Is it possible that a variable goes outside its meaningful range?		Ada compiler will check
7	For expressions with more than one operator, is the order of computation and precedence of operators correct?	✓	
8	Are there any invalid uses of integer arithmetic, particularly divisions? Note that *[2x(I/2)]* may not be equal to *I*.	✓	
9	Are there any computations on nonarithmetic variables?	✓	
	Comparison defects		
1	Are there any comparisons between variables having incompatible data types?	✓	
2	Are there any mixed-mode comparisons or comparisons between variables of different length?	✓	
3	Are the comparison operators correct?	✓	
4	Does the Boolean expression state what it is supposed to state? Programmers often make mistakes when writing Boolean expressions involving and/or.	✓	
5	Is the precedence or evaluation order of the Boolean expressions correct?	✓	
6	Do the operands of a Boolean expression have logical values (0 or 1)?	✓	
7	Are there any comparisons of equality between two floating point numbers? Note that *[10.0 x 0.1]* is seldom equal to 1.0.	✓	
	Control flow defects		
1	Does every loop eventually terminate? Devise an informal proof or arguments showing that each loop will terminate.	✓	
2	Does the program have any goto statements? If yes, can you eliminate them?	✓	
3	Is it possible that a loop will never be executed because the entry condition is false?	✓	

[EBE94] Ebenau, R.G. and Strauss, S.H. *Software Inspection Process.* McGraw-Hill, New York, 1994.

#	Checklist Question	Used	Why Rejected
4	Are there any off-by-one defects (i.e., more than one or fewer than one iteration)?	✓	
5	For each switch statement, does it have a default branch?	✓	
6	Are there any nonexhaustive decisions?	✓	
	Interface defects		
1	Is the number of formal parameters (in a called routine) equal to the number of actual parameters (in the calling routine)? Are their orders correct?		Ada compiler will check
2	Do the attributes (e.g., type and length) of each formal parameter match the attributes of the corresponding actual parameters?		Ada compiler will check
3	Does the unit system of each formal parameter match that of the corresponding actual parameters?	✓	
4	Does a function modify the value of a parameter which is intended to be an input value?	✓	
5	Do global variables have the same definitions and attributes in all functions referencing them?	✓	
	Input/output defects		
1	Have all fines been opened before use?	✓	
2	Are end-of-file (EOF) conditions detected and handled correctly?	✓	
3	Are end-of-line conditions detected and handled correctly?	✓	
4	Do the format specifications match the information in the input/output (I/O) statement? For example does the program expect an integer while the input is a character?		Not appropriate for Ada
5	Are there spelling or grammatical errors in the printed or displayed output?	✓	
6	Does the program check the validity of its input?	✓	
	Missing Code		
	The last and most common and serious defect in the program is missing code, i.e., when programs do not check a certain condition(s) or do not implement a certain function(s).	✓	

[FAG76] Fagan, M.E. Design and code inspection to reduce errors in program development. *IBM Systems Journal*, Vol. 15, No. 3, 1976, pp. 182-211.

#	Checklist Question	Used	Why Rejected
	I_1 Logic		
	Missing		
1	Are all constants defined?		Not appropriate for Ada
2	Are all unique values explicitly tested on input parameters?		Not appropriate for Ada
3	Are values stored after they are calculated?		Not appropriate for Ada
4	Are all defaults checked explicitly tested on input parameters?		Not appropriate for Ada
5	If character strings are created are they complete, are all delimiters shown?	✓	
6	If a keyword has many unique values, are they all checked?		Not appropriate for Ada
7	If a queue is being manipulated, can the execution be interrupted; if so, is queue protected by a locking structure; can queue be destroyed over an interrupt?		Not appropriate for Ada
8	Are register being restored on exits?		Not appropriate for Ada
9	In queuing/dequeuing should any value be decremented/incremented?		Not appropriate for Ada
10	Are all keywords tested in macro?		Not appropriate for Ada
11	Are all keyword related parameters tested in service routine?		Not appropriate for Ada
12	Are queues being held in isolation so that subsequent interrupting requestors are receiving spurious returns regarding the held queue?		Not appropriate for Ada
13	Should any registers be saved on entry?		Not appropriate for Ada
14	Are all increment counts properly initialized (0 or 1)?		Not appropriate for Ada
	Wrong		
1	Are absolutes shown where there should be symbolics?		Not appropriate for Ada
2	On comparison of two bytes, should all bits be compared?		Not appropriate for Ada
3	On built data strings, should they be character or hex?		Not appropriate for Ada
4	Are internal variables unique or confusing if concatenated?		Not appropriate for Ada
	Extra		
1	Are all blocks shown in design necessary or are they extraneous?		Design not available

[NASA94] Angellatta, R. et. al. Software Verification Plan for GCS, NASA Langley Research Center, Hampton, VA, December 1994.

#	Checklist Question	Used	Why Rejected
	Data Usage		
1	Are COMMON BLOCKS labeled with the same names as the global data stores defined in GCS Data Requirements Dictionary Part II?		Not appropriate for Ada; project specific
2	Do the variables in the COMMON BLOCKS use the same names and order as the variables in the global data stores defined in the GCS Data Requirements Dictionary Part II?		Not appropriate for Ada; project specific
3	Do the variables in the COMMON BLOCKS have the same data types, number of dimensions, and size of each dimension as specified in the GCS Data Requirements Dictionary Part I?		Not appropriate for Ada; project specific
4	If the code includes variables in addition to those defined in the global data stores in the GCS Data Requirements Dictionary Part II, are they defined, initialized, and used only within the scope of a subframe?		Not appropriate for Ada; project specific
5	Are references to array subscripts expressed in column, row order?		Not general enough
6	Is array subscript usage within array bounds?		Ada compiler will check
7	Are constant values used only as constants and not as variables?	✓	
8	Are DO loop index variables used only within the loop?	✓	
9	Does the code maintain that same loop index within a loop?	✓	
	Structure		
1	Does the code comply with the software architectures from the design?		Design not available
2	Does the code avoid the use of GOTO statements?	✓	
3	Do all the code's statements perform a clear function?		Too subjective
4	Is the code void of any isolated or dead code segments?		Automated check
	Functions and Subroutines		
1	Does each unit have a single function, and is it clearly described?	✓	
2	Do actual and formal parameters agree in number, order, dimension, and data type?		Ada compiler will check
3	Are the functions of subroutine input and output parameters described?	✓	
4	Are all the parameters passed to a subroutine used?	✓	

[NASA94] Angellatta, R. et. al. Software Verification Plan for GCS, NASA Langley Research Center, Hampton, VA, December 1994.

#	Checklist Question	Used	Why Rejected
5	Do the functions and subroutines return data of the correct type?		Ada compiler will check
6	Is there a call to GCS_SIM_RENDEZVOUS before each subroutine?		Project specific
7	Are calls to GCS_SIM_RENDEZVOUS void of all parameters?		Project specific
8	Does the code avoid using system calls?		Portability is not a concern
	Traceability		
1	Does the code satisfy all the requirements in the Requirements Traceability Matrix including all derived requirements?		requirements not available
2	Do units map to a well-defined section in the Design?		design not available
	Logic		
1	Do logical conditions correctly use logical operators (.AND., .OR., .NOT.)?		Too vague
2	Do logical conditions correctly use relational operators (.GT., .GE., .LT., .LE, .EQ., .NE.)?		Too vague
3	Are all logical conditions included?		Too vague
4	Are comparisons of real variables to exact values avoided?	✓	
5	Is loop nesting correct?	✓	
6	Do loops have single exit and single entry points?	✓	
	Exceptional Conditions		
1	Is there code to detect the exceptional conditions in the Non-Functional section of the Requirements Traceability Matrix?		Project specific
2	If an exceptional condition is detected, does the code print the appropriate message to FORTRAN logical unit 6?		Project and FORTRAN specific
	Computations		
1	Are mixed type mathematical expressions avoided?	✓	
2	Do computations contain values with the same unit dimensions?	✓	
3	Does the code avoid assigning real expressions to integers causing truncation?	✓	
4	Are bit manipulations done correctly?	✓	

[NASA94]	**Angellatta, R. et. al. Software Verification Plan for GCS, NASA Langley Research Center, Hampton, VA, December 1994.**		
	Compliance with Standards		
1	Does the code follow basic structured programming techniques?	✓	
2	Does the software code and documentation comply with the approved code standards?		Too vague; standards not specified

[ONE88] **O'Neill, D. and Ingram, A.L. Software Inspections Tutorial. In *SEI Technical Review*, Software Engineering Institute, Pittsburgh, 1988.**

#	Checklist Question	Used	Why Rejected
	Completeness		
1	Has traceability been assessed?		Requirements not available
2	Has available tool assistance been applied in assessing traceability?		Requirements not available
3	Have all predecessor requirements been accounted for?		Requirements not available
4	Were any product fragments revealed not to have traceability to the predecessor requirements?		Requirements not available
5	What is the relationship of requirements to product component: one-to-one, many-to-one, one-to-many?		Not yes/no
	Correctness		
1	Are structured programming prime constructs used correctly:		*[redundancies removed]*
1a	All: is the function commentary satisfied for the sequence?		Commentary not available
1b	Loops: Does the loop terminate?	✓	
1c	Loops: Is a one-time loop acceptable?	✓	
1d	Loops: Are there discrete steps through the loop?	✓	
1e	Loops: Is the control variable not modified in the loop?	✓	
1f	Loops: Is the loop initialized and terminated properly?	✓	
1g	Case: Is the domain partitioned exclusively and exhaustively?	✓	
2	Are proper programs composed of multiple prime programs limited to single entry and single exit?	✓	
3	Are disciplined data structures used to manipulate and transform data?		Too subjective
4	Does the input domain span all legal values?		Too vague
5	Does the input domain span all possible values, with systematic exception handling for illegal values?	✓	
6	Does the output range span all legal values?		Too vague
7	For modules, do the state data span all legal values?		Too vague

[ONE88]	**O'Neill, D. and Ingram, A.L. Software Inspections Tutorial. In *SEI Technical Review*, Software Engineering Institute, Pittsburgh, 1988.**		
Style			
1	Are style conventions for block structuring defined and followed?		Too vague: conventions not specified
2	Are naming conventions defined and followed?		Too vague: conventions not specified
3	Are the semantics of the product component traceable to the requirements?		Too vague: conventions not specified
4	Are style conventions for commentary defined and followed?		Too vague: conventions not specified
5	Are style conventions for alignment, upper/lower case, and highlighting defined and followed?		Pretty printer can do this
6	Are templates used for repeating patterns?		Process information not available
Rules of Construction			
1	Is the interprocess communication protocol defined and followed?		Too vague: protocol not specified
2	Are architectural conventions defined and followed for tasking and concurrent operations?		Too vague: conventions not specified
3	Are architectural guidelines defined and followed for program unit construction?		Too vague: guidelines not specified
4	Are data representation conventions defined and followed?		Too vague: conventions not specified
Multiple Views			
1	Has the product component been assessed for execution considerations such as timing, memory use, input and output, initialization, and finite word effects?		Efficiency is not a concern
2	Has the product component been assessed for packaging considerations such as program unit construction, program generation process, and target operation, including scheduling and memory management?		Logistics is not a concern
3	Has the product component been assessed for construction considerations such as structured programming, module strength and coupling, programming language, operating system, and physical hardware interfaces?		Not part of code inspection process
4	Has the product component been assessed for satisfying the functional baseline?		Baseline not available
5	Has the product component been assessed for user interface considerations?		Usability is not a concern

[ONE88]	O'Neill, D. and Ingram, A.L. Software Inspections Tutorial. In *SEI Technical Review*, Software Engineering Institute, Pittsburgh, 1988.		
	Metrics		
1	Have the source lines of code been recorded?		Metrics not appropriate for checklist
2	Has the project productivity been assessed?		Metrics not appropriate for checklist
3	Has the programmer productivity been assessed?		Metrics not appropriate for checklist
4	Have the errors per thousand lines of source code been recorded?		Metrics not appropriate for checklist
5	Has the product complexity been assessed?		Metrics not appropriate for checklist
6	Has the mean time to failure been assessed?		Metrics not appropriate for checklist
7	Has computer resource loading for CPU, memory, and I/O been assessed?		Metrics not appropriate for checklist
	Technology		
1	Is systematic programming used, including structured programming primes, proper programs, and disciplined data structures?		Redundant
2	Is systematic design used to produce string modules, loosely coupled?		Too subjective
3	Does the product component utilize information hiding, separation of concerns, and localization?		Too subjective
4	Does the product component utilize abstraction and encapsulation?		Too subjective
5	Is the product component understandable?		Too subjective
6	Is the product component maintainable and adaptable?		Maintainability and Adaptability not a concern

[SQF95] ***Software Quality Framework as Implemented in QUES Release 1.5*, Software Productivity Solutions, Inc., Indialantic, FL, February 1995.**

Identifier	Question	Answer Type	Used	Why Rejected
AC.1.05.e	During execution, are all outputs within the specified accuracy tolerances?	Y/N/NA	✓	
AM.1.05.e	When an error condition is detected, is its resolution determined by the calling CSU?	Y/N/NA	✓	
AM.2.01.e	Are error tolerances specified for all particular external input data (e.g., range of numerical values, legal combinations of alphanumeric values)?	Y/N/NA	✓	
AM.2.03.e	Are all applicable external inputs checked with respect to specified ranges before use?	Y/N/NA		Related to AM.1.05.e
AM.2.04.e	Are all applicable external inputs checked with respect to illegal combinations and conflicting requests prior to use?	Y/N/NA		Related to AM.1.05.e
AM.2.05.e	Are all detected errors with respect to applicable external inputs reported before processing begins?	Y/N/NA		Related to AM.1.05.e
AM.2.06.e	Are all applicable external inputs checked for reasonableness before processing begins?	Y/N/NA		Related to AM.1.05.e
AM.3.01.e	Is recovery provided for all computational failures within the CSU?	Y/N/NA		Related to AM.1.05.e
AM.3.02.e	Are all critical (i.e., supporting a mission-critical capability) loop and index parameters checked by explicit checks in the code or by features of the Ada language for out-of-range values before use?	Y/N/NA		Automatic for Ada
AM.3.03.e	Are all critical (i.e., supporting a mission-critical capability) subscript values checked for out-of-range values before use?	Y/N/NA		Automatic for Ada
AM.3.04.e	Are all critical (i.e., supporting a mission-critical capability) outputs checked for reasonable values before final outputting?	Y/N/NA	✓	
AM.4.01.e	Is recovery made (e.g., exception handlers or other means) from all detected hardware faults (e.g., arithmetic faults, hardware failure, clock interrupt)?	Y/N/NA		Related to AM.1.05.e
AP.1.01.e	Is the CSU free from references to the database management scheme (e.g., all data calls for data base information are processed through an executive)?	Y/N/NA		Not applicable to Correctness or Understandability

[SQF95]	***Software Quality Framework as Implemented in QUES Release 1.5*, Software Productivity Solutions, Inc., Indialantic, FL, February 1995.**			
AP.2.01.e	How many unique parameters are in the unit?	__/NA		Not meaningful by itself
AP.2.02.e	How many global variables are referenced by the unit? (a global variable is defined as a variable declared or visible in a unit other than that containing the code referencing the variable)?	__/NA	✓	
AP.2.03.e	Do the comments for global data in the CSU explain where the data is derived, the data's composition, and how the data is used?	Y/N/NA	✓	
AP.2.04.e	Does the unit contain comments for all parameter I/O and local variables describing each data item's composition and use?	Y/N/NA	✓	
AP.3.01.e	Is the unit free from computer architecture references?	Y/N/NA		Not applicable to Correctness or Understandability
AP.3.03.e	How many lines of code are there in the unit, excluding comments and blank lines?	__/NA		Not meaningful by itself
AP.3.04.e	How many non-HOL lines of code are there in the unit, excluding comment and blank lines? (For example, Assembly Language, Intermediate Code.)	__/NA	✓	
AP.4.01.e	Is the unit free from microcode instruction statements?	Y/N/NA		Not applicable to Correctness or Understandability
AT.1.01.e	Are all array bounds, upper and lower (e.g., index constraints) defined parametrically?	Y/N/NA		Not applicable to Correctness or Understandability
AT.2.01.e	Are all accuracy, convergence, timing attributes, and timing limitations defined parametrically for this CSU?	Y/N/NA		Not applicable to Correctness or Understandability
AT.2.02.e	Are tables used in a manner which would ease the task of changing or expanding capabilities?	Y/N/NA		Not applicable to Correctness or Understandability
AU.1.03.e	How many lines of source code are there to handle hardware/device interface protocol?	__/NA		Not meaningful by itself
AU.1.05.e	Is the CSU required to perform hardware/device interface protocol?	Y/N/NA		Not applicable to Correctness or Understandability
CL.1.03.e	Does the CSU receive input from other systems?	Y/N/NA		Not applicable to Correctness or Understandability
CL.1.04.e	Does the CSU transmit data to other systems?	Y/N/NA		Not applicable to Correctness or Understandability

[SQF95]	***Software Quality Framework as Implemented in QUES Release 1.5*, Software Productivity Solutions, Inc., Indialantic, FL, February 1995.**			
CL.2.02.e	Do all data representations and translations between representations of data comply with the established standard?	Y/N/NA		Too vague: standard not specified
CL.2.03.e	Does the unit perform data translation between representations of data?	Y/N/NA		Not applicable to Correctness or Understandability
CP.1.01.e	Are the inputs, processing, and outputs of the CSU specified?	Y/N/NA		Redundant
CP.1.02.e	How many data items are identified?	__/NA		Not meaningful by itself
CP.1.03.e	How many identified data items are defined (documented with regard to their source, meaning and format)?	__/NA	✓	
CP.1.04.e	How many identified data items are defined, computed, or obtained from an external source? (For example, referencing: global data with preassigned values, input parameters with preassigned values.)	__/NA		Not meaningful by itself
CP.1.05.e	How many data items are referenced?	__/NA		Not meaningful by itself
CP.1.06.e	How many defined data items are referenced?	__/NA	✓	
CP.1.10.e	Are all conditions and alternative processing options defined for each decision point in the CSU?	Y/N/NA		Excessive
CP.1.11.e	Are all parameters in the argument list used in the unit?	Y/N/NA	✓	
CS.1.04.e	Does the calling sequence comply with the established standard?	Y/N/NA		Too vague: standard not specified
CS.1.05.e	Does the CSU comply with the standards established for the calling sequence protocol between CSUs?	Y/N/NA		Too vague: standard not specified
CS.1.08.e	Does the CSU comply with the established standard for the external I/O protocol and format for all CSUs?	Y/N/NA		Too vague: standard not specified
CS.1.11.e	Does the handling of errors comply with the established standard?	Y/N/NA		Too vague: standard not specified
CS.2.05.e	Do the data names in this CSU comply with the established standard?	Y/N/NA		Too vague: standard not specified
CS.2.08.e	Do the definitions and uses of global variables comply with the established standard?	Y/N/NA		Too vague: standard not specified

[SQF95]	***Software Quality Framework as Implemented in QUES Release 1.5*, Software Productivity Solutions, Inc., Indialantic, FL, February 1995.**			
CS.2.14.e	Do all references to the same data use single unique names?	Y/N/NA	✓	
EP.1.01.e	Is the unit optimized for processing efficiency?	Y/N/NA		Not applicable to Correctness or Understandability
EP.1.03.e	Is the CSU optimized for processing efficiency (i.e., compiled with an optimizing compiler or coded in assembly language)?	Y/N/NA		Not applicable to Correctness or Understandability
EP.1.05.e	How many loops are in the unit (WHILE loops, REPEAT UNTIL loops, and iteration loops)?	__/NA		Not meaningful by itself
EP.1.06.e	How many loops contain non-loop dependent statements? (For example, initializing or calculating a variable which is not related to any values which change within the loop.)	__/NA		Not applicable to Correctness or Understandability
EP.1.07.e	How many instances are there of 2 or more operations in an expression (compound expression)?	__/NA		Not meaningful by itself
EP.1.08.e	How many compound expressions are recalculated needlessly (no variables in the expression have been reassigned values.)?	__/NA		Not applicable to Correctness or Understandability
EP.1.10.e	How many instances of bit/byte packing/unpacking are performed (i.e., use of the pragma PACK in Ada.)?	__/NA		Not meaningful by itself
EP.1.11.e	How many instances of bit/byte packing/unpacking are performed needlessly in a loop (i.e., could be performed outside the loop)? (Guaranteed score of 0 of pragma PACK is used in Ada.)	__/NA		Not applicable to Correctness or Understandability
EP.2.03.e	How many arithmetic expressions in the unit?	__/NA		Not meaningful by itself
EP.2.04.e	How many arithmetic expressions with different sized components in the same expression? (For example, byte/word/double word.)	__/NA		Not meaningful by itself
EP.2.05.e	How many arithmetic expressions with mixed data types in the same expression? (For example, integer/real/boolean/literal.)	__/NA	✓	
EP.2.06.e	How many data items are in the unit (arrays, constants, variables, etc.)?	__/NA		Not meaningful by itself
EP.2.07.e	How many data items are modified in the unit?	__/NA		Not meaningful by itself
ES.1.08.e	Are there any data packing operations in the unit?	Y/N/NA		Not applicable to Correctness or Understandability

[SQF95]	***Software Quality Framework as Implemented in QUES Release 1.5*, Software Productivity Solutions, Inc., Indialantic, FL, February 1995.**			
ES.1.09.e	Is the CSU optimized for storage efficiency (i.e., compiled with an optimizing compiler or coded in assembly language)?	Y/N/NA		Not applicable to Correctness or Understandability
FS.1.01.e	Does this CSU perform a single capability?	Y/N/NA		Not applicable to Correctness or Understandability
FS.1.02.e	Is a description of the capability(s) performed provided in the CSU's comments?	Y/N/NA		Not applicable to Correctness or Understandability
GE.2.01.e	Are the following processing categories mixed in this CSU: External Input, External Output, or Algorithmic Processing?	Y/N/NA		Not applicable to Correctness or Understandability
GE.2.03.e	Is this CSU free from machine-dependent operations (e.g., no representation clauses, compiler predefined pragmas, or use of system-defined types)?	Y/N/NA		Not applicable to Correctness or Understandability
GE.2.04.e	Can the volume of data processed by the unit be changed without effecting the executable code? (For example, data volume limits are parameterized.)	Y/N/NA		Not applicable to Correctness or Understandability
GE.2.05.e	Can the range of data input be changed without effecting the executable code? (For example, no error tolerances are specified, no range-tests or reasonableness checks are performed.)	Y/N/NA		Not applicable to Correctness or Understandability
ID.1.01.e	Is a standard subset of the implementation language used?	Y/N/NA		Not applicable for Ada
ID.1.03.e	How many references are there to system library subprograms, utilities, or other system provided facilities?	__/NA		Not applicable to Correctness or Understandability
ID.1.05.e	Is the unit free from nonstandard constructs of the implementation language(s)?	Y/N/NA		Not applicable for Ada
ID.2.03.e	Does the unit perform external input or output?	Y/N/NA		Not applicable to Correctness or Understandability
ID.2.04.e	Does the CSU contain operations dependent on word or character size?	Y/N/NA		Not applicable to Correctness or Understandability
ID.2.05.e	Does the unit contain data elements representations that are machine dependent?	Y/N/NA		Not applicable to Correctness or Understandability
MO.1.02.e	Is the CSU coded and tested according to structured techniques? (For example, top-down implementation and testing.)	Y/N/NA	✓	Testing information not available
MO.1.03.e	Does this CSU have a single processing objective (i.e., all processing within the CSU is related to the same objective)?	Y/N/NA		Too subjective

[SQF95]	***Software Quality Framework as Implemented in QUES Release 1.5*, Software Productivity Solutions, Inc., Indialantic, FL, February 1995.**			
MO.1.04.e	Are the lines of source code for this unit (excluding comment lines and blank lines) 100 lines or less?	Y/N/NA		Not generally accepted size threshold
MO.1.05.e	How many unique parameters are in the unit?	__/NA		Not meaningful by itself
MO.1.06.e	How many calling sequence parameters are control variables (select an operating mode or submode in the unit, direct the sequential flow, or otherwise directly influence the capability of the unit)?	__/NA		Not meaningful by itself
MO.1.07.e	Is all input data passed into the unit through calling sequence parameters (i.e., no data is input through global areas or input statements)?	Y/N/NA		Redundant with AP.2.02.e
MO.1.08.e	Is output data passed back to the calling unit? (For example, through calling sequence parameters or global areas.)	Y/N/NA		Not meaningful by itself
MO.1.09.e	Is control always returned to the calling unit when execution is completed?	Y/N/NA		Not required in all cases; too generalized
MO.1.10.e	Is temporary storage (work space reserved for intermediate or partial results) used only by this CSU during execution (i.e., not shared with other CSUs)?	Y/N/NA		Not applicable to Correctness or Understandability
MO.2.05.e	What is the cohesion value of this CSU?	__/NA		Not meaningful by itself
SD.1.02.e	How many comment lines are there in the unit?	__/NA		Not meaningful by itself
SD.1.03.e	How many lines of source code with embedded comments?	__/NA		Not meaningful by itself
SD.2.01.e	Does the CSU's prologue contain all the information in accordance with the established standard?	Y/N/NA	✓	
SD.2.02.e	Is the identification and placement of all comments in the CSU in accordance with the established standard?	Y/N/NA		Too vague: standard not specified
SD.2.03.e	Are all decision points and transfers of control commented in the CSU?	Y/N/NA		Excessive
SD.2.04.e	Is all machine-dependent code commented in the CSU?	Y/N/NA		Not applicable to Ada
SD.2.05.e	Are all nonstandard HOL statements commented in the CSU?	Y/N/NA		Not applicable to Ada

[SQF95]	***Software Quality Framework as Implemented in QUES Release 1.5*, Software Productivity Solutions, Inc., Indialantic, FL, February 1995.**		
SD.2.06.e	Are the attributes (usage, properties, units of measure) of all declared variables described by comments?	Y/N/NA	Excessive (Ada should be self-documenting)
SD.2.07.e	Do all the comments related to operations in the CSU describe the purpose or intent of the operation? (For example, comment states "INCREMENT TABLE LOOK-UP INDEX", rather than "INCREMENT A BY 1".)	Y/N/NA	Excessive (Ada should be self-documenting)
SD.2.08.e	Are the range of values and default conditions associated with all input parameters described by comment?	Y/N/NA	Related to AP.2.04.e
SD.3.01.e	Is the unit coded using only a higher order language?	Y/N/NA	Not applicable to Ada
SD.3.02.e	Are all variable names in the CSU descriptive of the physical or functional property they represent? (For example, variable names "XCOORD, YCOORD" rather than "A1, A2".)	Y/N/NA	Excessive (Ada should be self-documenting)
SD.3.03.e	Is all the code in the CSU logically blocked and indented?	Y/N/NA	Not applicable: pretty printer can do this
SD.3.04.e	How many lines of code are there in the unit with more than one statement?	__/NA	Not meaningful by itself
SD.3.05.e	How many continuation lines of code are there in the unit?	__/NA	Not meaningful by itself
SD.3.07.e	Is the CSU structured in the standard established format?	Y/N/NA	Not applicable: pretty printer can do this
SD.3.08.e	Are all language keywords in the unit used only with their predefined meaning? (For example, no keywords are also used as variable names.)	Y/N/NA	Not applicable to Ada
SI.1.02.e	Is the unit independent of the source of input and destination of output?	Y/N/NA	Too subjective
SI.1.03.e	Is the CSU independent of knowledge of prior processing?	Y/N/NA	Too subjective
SI.1.04.e	Does the CSU description/prologue include input, output, processing, and limitations?	Y/N/NA	Redundant
SI.1.05.e	How many non-tasking entrances into the unit?	__/NA	Not meaningful by itself
SI.1.06.e	How many non-tasking exits from the unit?	__/NA	Not meaningful by itself
SI.1.07.e	How many unique data items are in common blocks in this CSU?	__/NA	Not applicable to Ada
SI.1.08.e	How many unique common blocks in this CSU?	__/NA	Not meaningful by itself

[SQF95]	***Software Quality Framework as Implemented in QUES Release 1.5*, Software Productivity Solutions, Inc., Indialantic, FL, February 1995.**			
SI.1.11.e	Does this CSU description identify all interfacing CSUs and interfacing hardware?	Y/N/NA		Redundant
SI.3.01.e	How many conditional branch statements are there in the unit? (For example, IF, WHILE, REPEAT, DO/FOR loop, CASE.)	__/NA		Not meaningful by itself
SI.3.02.e	How many unconditional branch statements are there in the unit? (For example, GOTO, CALL, RETURN.)	__/NA		Not meaningful by itself
SI.4.01.e	Is the unit's flow of control from top to bottom (i.e., control does not erratically jump)?	Y/N/NA		Not applicable to Ada
SI.4.03.e	How many negative boolean and compound boolean expressions are used in the unit?	__/NA	✓	
SI.4.05.e	How many loops have unnatural exits? (For example, gotos exiting out of loops, return statements.)	__/NA		Too subjective
SI.4.06.e	How many iteration loops are used in the unit (DO/FOR loops)?	__/NA		Not meaningful by itself
SI.4.07.e	In how many iteration loops are indices modified to alter the fundamental processing of the loop? (Guaranteed value of 0 for Ada.)	__/NA		Not applicable to Ada
SI.4.08.e	Is the CSU free from all self modification of code? (For example, the unit doesn't alter instructions, overlays of code, etc.)	Y/N/NA		Not applicable to Ada
SI.4.09.e	How many statement labels are used in the unit, excluding labels for format statements?	__/NA		Not meaningful by itself
SI.4.10.e	What is the maximum nesting level in the unit?	__/NA		Not meaningful by itself
SI.4.11.e	How many total branches (conditional and unconditional) are used in the unit?	__/NA		Not meaningful by itself
SI.4.12.e	How many data declaration statements are there in the unit?	__/NA		Not meaningful by itself
SI.4.13.e	How many data manipulation statements are there in the unit?	__/NA		Not meaningful by itself
SI.4.14.e	How many total data items (local and global) are used in the unit?	__/NA		Not meaningful by itself
SI.4.15.e	How many local data items are referenced locally in the unit? (For example, variables declared locally and value parameters.)	__/NA		Redundant with CP.1.06.e

[SQF95]	***Software Quality Framework as Implemented in QUES Release 1.5*, Software Productivity Solutions, Inc., Indialantic, FL, February 1995.**			
SI.4.16.e	Does each data item in the CSU have a single use? (For example, each array serves only one purpose.)	Y/N/NA		Not applicable to Ada
SI.4.17.e	Is this CSU coded according to the required programming standard?	Y/N/NA		Too vague: standard not specified
SI.5.01.e	How many data items are used as input to the unit?	__/NA		Not meaningful by itself
SI.5.02.e	How many data items are used for output by the unit?	__/NA		Not meaningful by itself
SI.5.03.e	How many parameters in the unit's calling sequence return output values?	__/NA		Not meaningful by itself
SI.5.04.e	Does the unit perform a single, non-divisible capability?	Y/N/NA		Too subjective
SI.6.01.e	How many unique operators are in the unit?	__/NA		Not meaningful by itself
SI.6.02.e	How many unique operands are in the unit?	__/NA		Not meaningful by itself
SI.6.03.e	How many total operands are in the unit?	__/NA		Not meaningful by itself
ST.1.01.e	How many data items are in this unit's interface (i.e., data items used to input or output data to the unit)?	__/NA		Not meaningful by itself
ST.1.02.e	How many global data items in this CSU's interface are not adequately commented (i.e., lack comments regarding the purpose, type, limitations of the data)?	__/NA		Redundant
ST.1.03.e	How many interface data items are in the unit with negative qualification logic? (For example, boolean values that return "true" upon failure rather than success.)	__/NA	✓	
ST.1.04.e	Is the CSU interface established solely by arguments in the calling sequence parameter list (i.e., is the CSU free from references to global data)?	Y/N/NA		Redundant
ST.1.05.e	Does the CSU modify the internal code or data of other CSUs?	Y/N/NA		Not applicable to Ada
ST.2.01.e	How many unique execution paths are in the unit, including those caused by a "raise" statement?	__/NA		Not meaningful by itself
ST.2.02.e	How many conditional branch statements are in the unit? (For example, IF, WHILE, EXIT WHEN, CASE.)	__/NA		Not meaningful by itself

[SQF95]	***Software Quality Framework as Implemented in QUES Release 1.5*****, Software Productivity Solutions, Inc., Indialantic, FL, February 1995.**		
ST.2.03.e	How many other units are called by this unit? (For example, calls to other functions, generics, units, and tasks.)	__/NA	Not meaningful by itself
ST.2.04.e	How many iteration loops are there in the unit? (For example, DO/FOR loops.)	__/NA	Not meaningful by itself
ST.2.05.e	Are there comments regarding the CSUs called by this CSU and the CSUs which call this CSU?	Y/N/NA	Redundant
ST.3.04.e	Is temporary storage (work space reserved for intermediate or partial results) used only by this CSU during execution (i.e., not shared with other CSUs)?	Y/N/NA	Not applicable to Correctness or Understandability
ST.3.05.e	Does the CSU mix the management of primary and secondary storage resources with the management of data within the storage areas? (For example, an executive CSU that allocates storage for process, and controls what data can be accessed during process execution?)	Y/N/NA	Not applicable to Correctness or Understandability
ST.4.05.e	Does this unit have a single entrance (all units calling this unit must enter at the same location)? (In Ada, all non-tasking units will result in a guaranteed "Y".)	Y/N/NA	Not applicable to Ada
ST.4.06.e	Does this CSU's communication with all interfacing CSUs pass only data parameters (i.e., does not pass any control elements)?	Y/N/NA	Not applicable to Correctness or Understandability
ST.5.01.e	Is the CSU free from unnecessarily recomputing the same value?	Y/N/NA	Not applicable to Correctness or Understandability
ST.5.02.e	Is the CSU free from statements which cannot be executed?	Y/N/NA	Checked with automated tool
ST.5.03.e	Is the meaning of each data item consistent throughout the CSU (i.e., the use associated with each data item does not change)?	Y/N/NA	Redundant with CS.2.14.e
ST.5.04.e	Is the CSU free from unnecessary intermediate data items?	Y/N/NA	Not applicable to Correctness or Understandability
VS.1.01.e	How many execution paths are there in the unit?	__/NA	Not meaningful by itself
VS.1.03.e	How many total input parameters are there in the unit?	__/NA	Not meaningful by itself

Code Inspection Checklist for Ada

Data Collection Form

Identifier	Question	Answer
	• Computational •	
C.01.U	For functions that perform computations, are accuracy tolerances documented?	Yes / No / NA
C.02.C	Do all computations use variables with consistent types, modes, and lengths? (e.g., no boolean variables in arithmetic expressions, or mixed integer and floating-point)?	Yes / No / NA
C.03.C	Are all expressions free from the possibility of an underflow or overflow exception?	Yes / No / NA
C.04.C	Are all expressions free from the possibility of a division by zero?	Yes / No / NA
C.05.C	Is the order of computation and precedence of operators correct in all expressions?	Yes / No / NA
C.06.C	Are all expressions free from invalid uses of integer arithmetic, particularly divisions?	Yes / No / NA
C.07.C	Are all computations free from non arithmetic variables?	Yes / No / NA
C.08.C	Are all comparisons between variables of compatible data types, modes, and lengths?	Yes / No / NA
C.09.C	Do all comparisons avoid equality comparison of floating-point variables?	Yes / No / NA
C.10.C	Is the code free from assignment of a real expression to an integer variable?	Yes / No / NA
C.11.C	Are all bit manipulations correct?	Yes / No / NA
	• Data •	
D.01.C	Are all data items referenced?	Yes / No / NA
D.02.U	Do all references to the same data use single unique names?	Yes / No / NA
D.03.C	Are all character strings complete and correct, including delimiters?	Yes / No / NA
D.04.C	Are illegal input values systematically handled?	Yes / No / NA
D.05.C	Are all variables set or initialized before referenced?	Yes / No / NA
D.06.C	Are all array indexes integers?	Yes / No / NA
D.07.C	For all references through pointer variables, is the referenced storage currently allocated?	Yes / No / NA
D.08.C	Are all storage areas free from alias names with different pointer variables?	Yes / No / NA
D.09.C	Are all variables correctly initialized?	Yes / No / NA
D.10.C	Are all variables assigned to the correct length, type, storage class and range?	Yes / No / NA

D.11.U	Is the code free from variables with similar names (e.g., VOLT and VOLTS)	Yes / No / NA
D.12.C	Are all indexes properly initialized?	Yes / No / NA
D.13.U	Are all data declarations commented?	Yes / No / NA
D.14.U	Are all data names descriptive enough?	Yes / No / NA
D.15.C	Are constant values used only as constants and not as variables?	Yes / No / NA
D.16.C	For all arrays, is the attribute 'RANGE' used instead of numeric literals?	Yes / No / NA
D.17.U	Are error tolerances documented for all external input data?	Yes / No / NA

• Interface •

I.01.C	Are all propagated exceptions declared as visible and documented?	Yes / No / NA
I.02.C	Are all propagated exceptions handled (not raised) by the calling unit?	Yes / No / NA
I.03.C	Are reasonable ranges declared for all output values?	Yes / No / NA
I.04.C	For all global variables, is their use justified, and are they documented?	Yes / No / NA
I.05.U	Are all subprogram parameters modes shown and usage described via comments?	Yes / No / NA
I.06.U	Does the prologue document all side effects, such as propagated exceptions?	Yes / No / NA
I.07.U	Are there any interface data items with negative qualification logic (e.g., boolean values that return "true" upon failure rather than success)?	Yes / No / NA
I.08.C	Do all units systems of formal parameters match actual parameters (such as degrees vs. radians, or miles per hour vs. feet per second)?	Yes / No / NA
I.09.C	Are all functions free from modification of input parameters?	Yes / No / NA
I.10.C	Are global variables consistently used in all references?	Yes / No / NA
I.11.C	Are files opened before use and closed when finished?	Yes / No / NA
I.12.C	Are all input parameter variables referenced? Are all output values assigned?	Yes / No / NA
I.13.U	Does each unit have a single function, and is it clearly described?	Yes / No / NA
I.14.C	Are all functions free from side effects?	Yes / No / NA
I.15.C	Is there a single entry and a single exit?	Yes / No / NA

• Logic •

L.01.C	Are all negative boolean and compound boolean expressions correct?	Yes / No / NA
L.02.C	For all case statements, is the domain partitioned exclusively and exhaustively?	Yes / No / NA
L.03.C	Are all indexing operations and subscript references free from off-by-one defects?	Yes / No / NA
L.04.C	Are all comparison operators correct?	Yes / No / NA
L.05.C	Are all boolean expressions correct?	Yes / No / NA
L.06.C	Is the precedence or evaluation order of boolean expressions correct?	Yes / No / NA
L.07.C	Do the operands of boolean expressions have logical values (0 or 1)?	Yes / No / NA
L.08.C	Does every loop eventually terminate?	Yes / No / NA
L.09.C	Is the program free from goto statements?	Yes / No / NA
L.10.C	Are all loops free from off-by-one defects (i.e., more than one or fewer than one iteration)?	Yes / No / NA
L.11.C	Are all case statements free from others branches?	Yes / No / NA
L.12.C	Are all decisions exhaustive?	Yes / No / NA
L.13.C	Are end-of-file conditions detected and handled correctly?	Yes / No / NA
L.14.C	Are end-of-line conditions detected and handled correctly?	Yes / No / NA
L.15.C	Do processes occur in the correct sequence?	Yes / No / NA
L.16.C	Are all loops free from unnecessary statements?	Yes / No / NA
L.17.C	Are all loop limits correct?	Yes / No / NA
L.18.C	Are all branch conditions correct?	Yes / No / NA
L.19.C	Are loop index variables used only within the loop?	Yes / No / NA
L.20.C	Are all loops free from loop index modification?	Yes / No / NA
L.21.C	Is all loop nesting in the correct order?	Yes / No / NA
L.22.U	Do all loops have single exit and entry points?	Yes / No / NA
L.23.U	For all nested loops, are loops and loop exits labeled?	Yes / No / NA

• Other •

O.01.U	Is the prologue complete and correct?	Yes / No / NA
O.02.C	Are all printed or displayed messages free from grammatical or spelling errors?	Yes / No / NA
O.03.U	Does the code follow basic structured programming techniques?	Yes / No / NA
O.04.U	Are all assumptions documented?	Yes / No / NA
O.05.C	Is the code written only in Ada?	Yes / No / NA

List of Sources for the C++ Checklist in this Appendix:

[BAL92] Baldwin, John T. "An Abbreviated C++ Code Inspection Checklist." October 27, 1992. http: //www.ics.hawaii.edu/~johnson/FTR/Bib/Baldwin92.html.

[DST96] "C Code Review Checklist." http://dstc.qut.edu.au/~baker/www/sqg/C_Checklist.html.

[FAG96] Fagan, Michael. "OLP Software Inspection." http://www-ols.fnal.gov:8000/ols/www/inspection.html#focus.

[FAU94] Faure, John. "Draft Standards for C++ Usage." Internal corporate software development standard for Software Productivity Solutions, Inc. September 8, 1994.

[GER95] Gerisch, Margaret. "Code Review Checklist." http://www.oswego.edu/~more/html/checklist2.html.

[HUM95] Humphrey, Watts. *A Discipline For Software Engineering*. SEI Series in Software Engineering. 1995.

[KOE92] Koenig, Andrew. "Checklist for Class Authors." *The C++ Journal*. Volume 2. No. 1. 1992.

[KOE95] Koenig, Andrew. "Working Paper for Draft Proposed International Standard for Information System—Programming Language C++." Doc No. X3J16/95-0185 WG21/NO785. September 26, 1995.

[MCC96] McCabe, Thomas. "McCabe OO Tool." Presentation materials from On-Site Tutorial. 1996.

[POT94] Potts, Stephen and Timothy S. Monk. *Borland C++ By Example*. Que Corporation. ISBN: 1-56529-756-3. 1994.

[SOF95] Software Productivity Solutions, Inc. "Task Area: Software Quality Framework." Interim Technical Report. Data & Analysis Center for Software, Subcontract No. P48124 under Prime Contract No. F30602-92-C-0158. October, 1995.

[SOF96] Software Productivity Solutions, Inc. "Certification of Reusable Software Components: Volume 5, Certification Field Trial." Contract No. F30602-94-C-0024. United States Air Force, Rome Laboratory. Rome, NY. June 24, 1996.

[VAN95] Ger van Diepen. "General C++ Coding Standard at the NFRA." http://www.nfra.nl/~qvd/seg/CppStdDoc.html.

Code Inspection Checklist for C++

Data Collection Form

Identifier	*Question*	Answer

• Computational •

C.01.U	*For functions that perform computations, are accuracy tolerances documented?* For functions that perform computations, are accuracy tolerances documented for variable types that hold data?	Yes / No / NA
C.02.C	*Do all computations use variables with consistent types, modes, and lengths (e.g., no boolean variables in arithmetic expressions, or mixed integer and floating-point)?* Do all computations use variables with consistent types and/or type casting, values, and lengths? (i.e., no boolean variables in arithmetic expressions) If variable types are mixed, are expected outcomes anticipated and external to the program block?	Yes / No / NA
C.03.C	*Are all expressions free from the possibility of an underflow or overflow exception?*	Yes / No / NA
C.04.C	*Are all expressions free from the possibility of a division by zero?*	Yes / No / NA
C.05.C	*Is the order of computation and precedence of operators correct in all expressions?*	Yes / No / NA
C.06.C	*Are all expressions free from invalid uses of integer arithmetic, particularly divisions?*	Yes / No / NA
C.07.C	*Are all computations free from non-arithmetic variables?*	Yes / No / NA
C.08.C	*Are all comparisons between variables of compatible data types, modes, and lengths?* Are all comparisons between variables of compatible data types, type cast data types, and lengths?	Yes / No / NA
C.09.C	Do all comparisons avoid equality comparison of floating-point variables?	Yes / No / NA
C.10.C	Is the code free from assignment of a real expression to an integer variable?	Yes / No / NA
C.11.C	*Are all bit manipulations correct?*	Yes / No / NA
C.12.C	Is the "%" modulus operator used correctly (i.e. not intended as a percentage)?	Yes / No / NA
C.13.C	Is the "/" division operator used to accommodate a discarded remainder?	Yes / No / NA
C.14.C	Are compound operators assigned correctly?	Yes / No / NA

• Data •

D.01.C	*Are all data items referenced?*	Yes / No / NA
D.02.U	*Do all references to the same data use single unique names?*	Yes / No / NA
D.03.C	*Are all character strings complete and correct, including delimiters?* Are all character strings and character arrays complete and correct, including delimiters (i.e., value is assigned and enough elements are reserved to hold entire character string and terminating null zero)?	Yes / No / NA
D.04.C	*Are illegal input values systematically handled?*	Yes / No / NA
D.05.C	*Are all variables set or initialized before referenced?*	Yes / No / NA
D.06.C	*Are all array indexes integers?*	Yes / No / NA
D.07.C	*For all references through pointer variables, is the referenced storage currently allocated?*	Yes / No / NA
~~D.08.C~~	~~*Are all storage areas free from alias names with different pointer variables?*~~	~~Yes / No / NA~~
D.09.C	*Are all variables correctly initialized?* Are all variable and constants correctly initialized?	Yes / No / NA
D.10.C	*Are all variables assigned to the correct length, type, storage class and range?* Are all variables and constants assigned to the correct length, type, sign, precision, and range?	Yes / No / NA
D.11.U	*Is the code free from variables with similar names (e.g., VOLT and VOLTS)?* Is the code free from variables and constants with similar names (e.g., VOLT and VOLTS)?	Yes / No / NA
D.12.C	*Are all indexes properly initialized?* Are all indexes properly initialized (i.e., start at zero)?	Yes / No / NA
D.13.U	*Are all data declarations commented?*	Yes / No / NA
D.14.U	*Are all data names descriptive enough?*	Yes / No / NA
D.15.C	*Are constant values declared as constants and not as variables?* Are constant values used as numbers, characters, words, or phrases?	Yes / No / NA
D.16.C	For all arrays or enumeration types, are ranges used for each data type instead of numeric literals?	Yes / No / NA
D.17.U	*Are error tolerances documented for all external input data?*	Yes / No / NA
D.18.U	Are variable names in lower case as is the customary convention?	Yes / No / NA
D.19.U	For object-oriented code, are the first letters of class names capitalized as is the customary convention?	Yes / No / NA

D.20.U	Are upper case letters used for "#define" directives as is the customary convention?	Yes / No / NA
D.21.U	Are "#define" statement used judiciously?	Yes / No / NA
D.22.C	Are assignment equals "=" and equals to "==" operators used correctly?	Yes / No / NA
D.23.C	Have assignment expressions been included in the same condition as the logical test?	Yes / No / NA
D.24.U	Are parenthesis used in the expressions of the "sizeof" operator (i.e., in "sizeof data", parentheses is optional, but it is good programming to include (); Are parenthesis used in the expressions of the "sizeof (data type) where parentheses are required?	Yes / No / NA
D.25.C	Are bitwise operators, bitwise shift, and compound bitwise shift used correctly (i.e., &, vertical bar, ^, ~, >>, <<, <<=, >>=)?	Yes / No / NA
D.26.C	For object-oriented components, do classes have any virtual functions? If so, is the destructor non-virtual?	Yes / No / NA
D.27.C	For object-oriented components, do classes have all three necessary copy-constructors, assignment operators, and destructors?	Yes / No / NA
D.28.C	For object-oriented components, do all structures and classes use the "." reference?	Yes / No / NA
D.29.C	Are all pointers initialized to "null", deleted only after "new", and new pointers deleted after use?	Yes / No / NA
D.30.C	Are names used within the declared scope?	Yes / No / NA
D.31.C	For object-oriented components, is each class declared and implemented in a single file (i.e., with the exception of helper classes packaged with the primary file)?	Yes / No / NA
D.32.C	Are function arguments free from variable argument lists (...) to avoid the inherently type-unsafe?	Yes / No / NA
D.33.U	Is multiple inheritance avoided?	Yes / No / NA
D.34.U	Are "return" types always provided, even if "void"?	Yes / No / NA
D.35.C	For object-oriented components, does every constructor initialize every data member in its class?	Yes / No / NA
D.36.C	For object-oriented components, do assignment operators correctly handle assigning an object to itself?	Yes / No / NA
D.37.C	Is "delete [ ]" used when deleting an array to determine the size of the array being deleted?	Yes / No / NA
D.38.U	For object-oriented components, are object fine grained?	Yes / No / NA

D.39.U	For object-oriented components, is the object encapsulated (i.e., highly related methods and data isolated)?	Yes / No / NA
D.40.U	For object-oriented components, is there low dependency between objects?	Yes / No / NA
D.41.U	For object-oriented components, do objects exhibit high fan in?	Yes / No / NA

• Interface •

~~I.01.C~~	~~*Are all propagated exceptions declared as visible and documented?*~~	~~Yes / No / NA~~
~~I.02.C~~	~~*Are all propagated exceptions handled (not raised) by the calling unit?*~~	~~Yes / No / NA~~
I.03.C	*Are reasonable ranges declared for all output values?*	Yes / No / NA
I.04.C	*For all global variables, is their use justified, and are they documented?*	Yes / No / NA
I.05.U	*Are all subprogram parameter modes shown and usage described via comments?* Are all subprogram parameter types shown and usage described via comments?	Yes / No / NA
I.06.U	*Does the prologue document all side effects, such as propagated exceptions?* Does the prologue document all side effects?	Yes / No / NA
I.07.U	*Are the interface data items free from negative qualification logic (e.g., boolean values that return "true" upon failure rather than success)?*	Yes / No / NA
I.08.C	*Do all units systems of formal parameters match actual parameters (such as degrees vs. radians, or miles per hour vs. feet per second)?*	Yes / No / NA
I.09.C	Are all functions free from modification of input parameters?	Yes / No / NA
I.10.C	*Are global variables consistently used in all references?*	Yes / No / NA
I.11.C	*Are files opened before use and closed when finished?* Are files opened immediately prior to access and closed as soon as done?	Yes / No / NA
I.12.C	*Are all input parameter variables referenced? Are all output values assigned?*	Yes / No / NA
I.13.U	*Does each unit have a single function, and is it clearly described?*	Yes / No / NA
I.14.C	*Are all functions free from side effects?*	Yes / No / NA
I.15.C	*Is there a single entry and a single exit?*	Yes / No / NA
I.16.C	Does the program and all its functions end with a return statement?	Yes / No / NA
I.17.C	Does each return have a closing brace (i.e., after the end of a block, the end of the main function [main ( )], and the end of the program?	Yes / No / NA
I.18.C	Are the widths and formats of numbers specified correctly for printing?	Yes / No / NA
I.19.C	Are the most frequently executed statements in a "switch" arranged at the top of the list to improve the efficiency of the code?	Yes / No / NA

I.20.C	If "ios::out" is used to open a file for writing (i.e., C++ creates the file), does it overwrite the filename that exists?	Yes / No / NA
I.21.U	Is code free from "non-standard" syntactic constructs such as unconventional preprocessor directives?	Yes / No / NA
I.22.C	Is passing objects by value, or by reference avoided (e.g., where implicit conversions result in member wise copying)? Are dynamically allocated application objects passed as pointers?	Yes / No / NA
I.23.C	To decrease performance overhead, are local variables created and assigned at once?	Yes / No / NA
I.24.C	Are files properly declared, opened, and closed?	Yes / No / NA
I.25.C	Is a file closed in the case of an error return?	Yes / No / NA
I.26.C	Are all "include" statements complete?	Yes / No / NA
I.27.C	Are "inline" functions used only when performance is needed?	Yes / No / NA
I.28.C	Are "new" and "delete" used to allocate and deallocate storage rather then "malloc" and "free" (i.e., which are type-unsafe)?	Yes / No / NA
I.29.C	Have timing, sizing, and throughput been addressed?	Yes / No / NA

• Logic •

L.01.C	*Are all negative boolean and compound boolean expressions correct?*	Yes / No / NA
L.02.C	*For all case statements, is the domain partitioned exclusively and exhaustively?* For all "switch" statements, is the domain partitioned exclusively and exhaustively?	Yes / No / NA
L.03.C	*Are all indexing operations and subscript references free from off-by-one defects?*	Yes / No / NA
L.04.C	*Are all comparison operators correct?*	Yes / No / NA
L.05.C	*Are all boolean expressions correct?*	Yes / No / NA
L.06.C	*Is the precedence or evaluation order of boolean expressions correct?*	Yes / No / NA
L.07.C	Do the operands of boolean expressions have logical values (0 or 1) or a non zero value which is interpreted as true?	Yes / No / NA
L.08.C	*Does every loop eventually terminate?*	Yes / No / NA
L.09.C	*Is the program free from goto statements?* Are "gotos" used judiciously or can other code be substituted?	
L.10.C	*Are all loops free from off-by-one defects (i.e., more than one or fewer than one iteration)?*	Yes / No / NA
L.11.C	Are all switch statements free from "others" branches?	Yes / No / NA
L.12.C	*Are all decisions exhaustive?*	Yes / No / NA

L.13.C	*Are end-of-file conditions detected and handled correctly?*	Yes / No / NA
L.14.C	*Are end-of-line conditions detected and handled correctly?*	Yes / No / NA
L.15.C	*Do processes occur in the correct sequence?*	Yes / No / NA
L.16.C	*Are all loops free from unnecessary statements?*	Yes / No / NA
L.17.C	*Are all loop limits correct?*	Yes / No / NA
L.18.C	*Are all branch conditions correct?*	Yes / No / NA
L.19.C	Are loop index variables used only within the loop?	Yes / No / NA
L.20.C	Are all loops free from loop index modification?	Yes / No / NA
L.21.C	*Is all loop nesting in the correct order?*	Yes / No / NA
L.22.U	Do all loops have single exit and entry points?	Yes / No / NA
L.23.U	For all nested loops, are loops and loop exits labeled?	Yes / No / NA
L.24.C	Is the ternary conditional operator “?:” used correctly?	Yes / No / NA
L.25.C	Are the increment and decrement operators properly used in postfix and prefix order?	Yes / No / NA
L.26.U	Do braces surround the body of a “for” and ”while” loop even though it only has one statement (i.e., exhibiting good programming practices)?	Yes / No / NA
L.27.U	Are the expected executions anticipated with “while”, “do while”, and “if while”, even though the code will compile?	Yes / No / NA
L.28.C	Are “exit (status)”, “break in case”, and “break and continue” used to correctly exit the program or exit the loop?	Yes / No / NA
L.29.C	Are counters initialized to zero and the increment operator (i.e., “++”) used appropriately?	Yes / No / NA
L.30.C	When “for” loops are used, is the intent for the condition to be tested at the top of the loop (i.e., is the condition ever “True” so that the loop executes)?	Yes / No / NA
L.31.C	Is redundancy eliminated in “for” loops for better efficiency?	Yes / No / NA
L.32.C	Do all “switch” statements contain a default branch to handle unexpected cases?	Yes / No / NA
L.33.C	Does logic handle bad input as well as good input?	Yes / No / NA

• Other •

O.01.U	*Is the descriptive prologue complete and correct?*	Yes / No / NA
O.02.C	*Are all printed or displayed messages free from grammatical or spelling errors?*	Yes / No / NA
O.03.U	*Does the code follow basic structured programming techniques?*	Yes / No / NA
O.04.U	*Are all assumptions documented?*	Yes / No / NA

O.05.C	*Is the code written only in Ada?* Is the code written only in C or C++?	Yes / No / NA
O.06.U	Is each variable declared on a single line to improve readability and maintainability?	Yes / No / NA
O.07.U	Does code contain mapping to parent documents, or functional specifications?	Yes / No / NA

Appendix C - Survey Data

This appendix contains the survey checklists and the results of interviews with survey participants.

Survey Checklists

Reuser

1) How would you describe your role in the organization?
   - Primary responsibilities
   - Official title
   - Length of time in this position
   - Formal education and training
   - Professional experience (assignments and application areas)
2) Describe your organization's interest and involvement in software reuse.
   - Mission or business objective
   - Standards and policies applicable to the software development process
   - Requirements for reuse
   - Number of developers
   - Platform support
3) Can you describe the application area(s) you work in?
   - Application-driven characteristics and quality requirements
   - Testing requirements and standards
   - Programming languages used
   - Software technologies used in the domain (e.g., databases, graphical user interfaces, communications systems)
4) Can you describe your current project?
   - Individual's development responsibilities
   - Resource constraints (time, funding, staff)
   - Software reused
5) Can you describe your organization's reusable asset collection?
   - Number of assets
   - Types of assets
   - Quality of assets
   - Frequency of individual's use
   - Number of assets individual has obtained and used

- Last asset obtained and reused
- Direct costs for obtaining and/or reusing assets
- Use of other repositories or library systems

6) Can you describe your search and retrieval process?
   - Searching process and mechanism
   - Process for determining asset functionality
   - Process for determining asset quality
   - Additional review performed after obtaining asset
   - Additional testing performed after obtaining asset
   - Special tools used in obtaining and/or reusing the asset
7) Can you describe what makes a "good" asset?
   - Minimum requirements
   - Additional capabilities or qualities desired in assets
   - Reasons for not reusing an asset
   - Successful reuse experiences
   - Unsuccessful reuse experiences
   - Satisfaction with assets provided
8) What would help you reuse more?
   - Additional or different assets
   - Higher quality assets
   - Additional testing or certification desired
   - Ability to pay for assets or services
   - Feedback individual has provided to appropriate staff
   - Plans for continued use of the asset collection
   - Would asset certification appeal to the reuser
   - Is there an incentive for reusing assets (does certification play a role)

Asset Developer

1) Please describe your organization and its interest and involvement in software reuse.
   - Mission or business objective
   - Commitment to specific reuse goals and objectives
   - Number of reusable asset developers
   - Number of asset reusers
2) Why do you develop reusable assets?
   - Part of a special reuse group within organization
   - Provides assets as a service to other external organizations (e.g., free or on cost-reimbursement basis)
   - Provides assets for fee to other external organizations
   - Building reusable assets for a particular project
   - Commercial supplier of assets
3) How would you describe your role in the organization?
   - Primary responsibilities
   - Official title
   - Length of time in this position
   - Formal education and training
   - Professional experience (assignments and application areas)
4) Can you describe the application area(s) you work in?
   - Application domains
   - Application-driven characteristics and quality requirements
   - Types of assets
   - Software technologies used in the domain (e.g., databases, graphical user interfaces, communications systems)
5) Can you describe the development of reusable assets?
   - What are the reuse goals
   - Is there an incentive for developing reusable assets
   - Process for identifying reuse needs or opportunities
   - Specific standards or guidelines applied to the development process
   - Process for determining and measuring quality requirements
   - Methods and tools used for testing
   - Collection of metrics during development
   - Availability of documentation on development process (can we get a copy)

- Procedure for making assets available for reuse
- Satisfaction with development process
- Identification of changes to be made in development process

6) Can you describe the automated environment you use in your work?
    - Special tools used to compose, generate, or synthesize assets
    - Programming languages used
    - Any other tools

7) Is there a lot of interaction with users of the assets?
    - Joint identification of requirements, architectures, interface design
    - Field testing of assets
    - Feedback on quality and functionality
    - Requests for fixes
    - Requests for additional testing
    - What increases the incentive for reusing assets

Librarian

1) Please describe your organization and its interest and involvement in software reuse.
   - Mission or business objective
   - Commitment to specific reuse goals and objectives
   - Number of reusable asset developers
   - Number of asset reusers
2) How would you describe your role in the organization?
   - Primary responsibilities
   - Official title
   - Length of time in this position
   - Formal education and training (approach this question carefully)
   - Professional experience (assignments and application areas)
3) Can you describe your typical workday?
   - Amount of time spent analyzing submitted assets and their associated information
   - Amount of time spent certifying/evaluating assets (if part of responsibility)
   - Amount of time spent cataloguing assets
   - Amount of time spent updating assets
4) Can you describe your staff?
   - Number of staff employed to maintain the collection of assets
   - Training provided to staff
   - Adequacy of staffing (numbers, skill levels, training)
   - Status of funding for staff (stable, increasing, decreasing)
   - Ability to add staff
5) Is there a lot of interaction required with individuals outside of your staff?
   - Method for acquiring new assets
   - Method for dealing with assets that don't meet requirements
   - Interaction with domain experts and/or asset developers
   - Interaction with certification engineers
   - Interaction with users
   - How are other individuals notified of the assets
6) Can you describe your collection of reusable assets?
   - Number of assets
   - Types of assets

- Application domains
- Application-driven characteristics and quality requirements
- Platform support
- Who provides the assets and why
- Standards and policies applicable to the software development process
- Frequency of new submittals
- Frequency of updates or modifications to assets
- Motivation for users to access and obtain assets from this collection
- Formal (written) description of collection (can we get a copy)

7) Can you describe your users?
   - Who are the users
   - Geographic distribution of user population
   - How many users
   - Frequency of new users
   - Collection of usage metrics
   - Availability of documentation on usage (can we get a copy)

8) Can you describe the supporting automation (e.g., library system) for the asset collection
   - Software used
   - Identification as commercial product, GFE, or custom product(s)
   - Capability for integration with other tools or systems
   - Availability of user documentation for supporting automation (can we get a copy)
   - Tools other than "library system" used to maintain collection

9) Can you describe your certification/evaluation process?
   - Minimum requirements for asset submittal
   - Availability of instructions for submitting assets (can we get a copy)
   - Method for determining level of certification required for an asset
   - Average effort spent certifying an asset
   - Availability of documented certification procedures (can we get a copy)
   - Adequacy of time spent on certification activities
   - User feedback received with regard to certification activities, results, or information
   - Identification of desired changes in process or supporting automation
   - Motivation for changing process and/or supporting automation

10) What kind of feedback have you received from users?
   - Satisfaction with quality and features of assets found
   - Services and/or information that users would like more of
11) How do you measure the successof your reuse effort?
   - Specific objectives that are measured
   - Mechanisms for measurement of success
   - What are the reuse goals
12) What kinds of changes, if any, might occur over the next three years?
   - Different types of assets
   - Different user populations
   - Different services and/or automation

Certification Engineer

1) Please describe your organization and its interest and involvement in software reuse.
   - Mission or business objective
   - Commitment to specific reuse goals and objectives
   - Number of reusable asset developers
   - Number of asset reusers
2) How would you describe your role in the organization?
   - Primary responsibilities
   - Official title
   - Length of time in this position
   - Formal education and training
   - Professional experience (assignments and application areas)
   - Specific training in certification, quality control, verification and validation
   - Is there a perceived need for certification
3) Can you describe your typical workday?
   - Amount of time spent analyzing submitted assets and their associated information
   - Amount of time spent determining how to certify assets
   - Amount of time spent certifying the assets
   - Amount of time documenting process and/or results
4) Is there a lot of interaction required with other individuals?
   - Method for dealing with assets that don't meet certification requirements
   - Interaction with librarian
   - Interaction with domain experts and/or asset developers
   - Interaction with users
   - How is the certification information relayed to other individuals
   - Where do the assets to certify come from
5) Can you describe the assets you certify?
   - Application domains
   - Application-driven characteristics and quality requirements
   - Types of assets
   - Volume of assets certified
   - Where in the software lifecycle is certification being performed

- What percentage of assets are being certified and how thorough is the certification

6) Can you describe the certification process you follow?
   - Minimum requirements for asset
   - Use (in certification) of information submitted with asset
   - Receipt and use of problem reports, field usage, or user feedback on asset quality
   - Method for determining level of certification required for an asset
   - Application-specific model or information used to set requirements for certifying an asset
   - Process defined such that different individuals performing same activity would receive same results
   - Availability of documented certification procedures
   - Testing and/or analysis techniques used
   - Average effort spent in certifying an asset
   - Description of any work product(s) that result from certifying an asset
   - Variation in process based on type of asset
   - Most recent asset certified and length of time required to certify it
   - Ability to change the process

7) Can you describe the automated environment you use in your work?
   - Activities that are automated
   - Software used
   - Use of special testing or measurement software
   - Use of software for managing asset collection
   - Activities that are not automated

8) Is the certification process satisfactory?
   - Adequacy of level of certification
   - Adequacy of time spent on certification activities
   - Adequate automation provided to support certification process
   - User feedback received with regard to certification activities, results, or information
   - Identification of desired changes in process or supporting automation
   - Motivation for changing process and/or supporting automation

Interviews

This section contains the interview summaries. Any information pertaining to the identity and company of each participant has been omitted.

Participant 1

This participant works in the domain of command and control system maintenance. He is a project manager in a corporation that performs rudimentary reuse. Software engineers, during development, ask other software engineers in their area for assets that meet certain requirements. There is no master list or repository of software assets. Reuse is very informal. To ensure that assets are reused, there is a checklist that the developers use during development. One item on this checklist requires that the developers look for assets prior to developing them. Other than this checklist, there is no incentive for reuse.

Each project has an Independent Testing Group (ITG) that handles asset certification and testing. Certification, to the ITG, means that an asset has been tested to meet its requirements. Prior to delivering a build to the customer the ITG is given the software along with a test plan. The ITG takes the software and tests it against its requirements using data based upon the developer's test plan. Every asset that is delivered to the customer, reused or new, is tested against its requirements. If an asset fails testing, it is tested with the programmer present. If it fails again, the customer is notified.

The ITG does not maintain test information nor does it keep track of the assets that have been tested after the project ends.

Participant 2

Participant 2 is the division point of contact for reuse in his division. He is currently setting up a domain independent reuse framework. The reuse framework will encompass a broad set of domains including law enforcement and weather. Since the framework is in its early stages, there are a lot of specifics that still need to be defined. He wants to use the same software maturity index that the DOD uses: that defined by the Defense Software Repository System . He is also setting up a software maturity process where a quality assurance team and a configuration control board will determine where an asset will fit in the software maturity index.

There is, in place, an automated domain independent library system from which reusers can browse and retrieve assets. Developers that submit assets will also submit verification that QA has been performed on the asset. The participant is setting up software development classes and software reuse classes for the software developers.

The participant thinks that accessibility to reusable components is more important than certification. A level of certification is important, but having potentially reusable assets in a location accessible to the software engineers is more important.

Participant 3

Participant 3 is working to apply Smalltalk technologies to conventional business systems. She is trying to apply the reusability that Smalltalk provides to their development efforts. Traditional CASE technology developed assets are taken and object-oriented wrappers are placed around them. She is examining the large classes of objects that can be found in the commercial market for reuse suitability.

An asset development team has been created to build reusable objects. The team is trying to understand platform independent object management. The asset development team is building objects within an already defined reuse framework.

Software development goes through the SEI levels of understanding: Approval To Proceed (ATP) 1 through 4. Most objects are currently at an ATP 2 level. Test cases are being defined so that regression testing can be performed, but they are finding it hard to do this.

She thinks that an object's maturity is an issue, but that problems are expected to arise during reuse, hence an object's maturity won't be a top issue. Although she does feel that there is a need for certification tools in the reuse market, that need is less than the need to understand how to reuse an asset.

To provide reuse information to the reuser, her reuse team is looking at ways to respond to reusers querying the library system for objects that perform a certain function. In addition, she is looking at ways to enable a reuser to quickly understand the behavior of an object.

Participant 4

Participant 4 is looking at ways to get reusable assets from existing code and then certify them. The existing assets are coming from legacy systems or previously developed software systems independent of domain. Complexity metrics are used to identify potentially reusable assets. Once an asset is identified, dependencies between the asset and its environment and other assets are removed. The asset is then documented and, eventually, a cataloguing system will be used to catalog the asset.

She is focusing on identifying assets whose outputs can be defined as a function of the asset inputs: output = f(input). This capability is key to their certification method. A formal specification is generated from the asset, and the correctness of the specification is tested. Once this is done, the functional correctness of the asset will be tested. Once the formal specification is defined, the asset can be certified. If the asset's behavior is consistent with its formal specification, it can be certified. This method is being developed internally and, currently, there is no documentation available.

Two levels of certification have been defined:

1) Certification checking that the formal specification and the asset are consistent

2) Certification using test cases on the asset to verify that it performs according to its function specification.

The formal specification includes only the asset's input and output parameters. The input set is defined as a certain *type*. The formal specification will map the set of outputs to all inputs of this *type*. Possible inputs outside of this *type* will not be mapped. The formal specification is used to raise the level of confidence that an asset does what it is supposed to do given inputs of the expected type. Information will be stored within the formal specification that will increase or decrease the level of confidence in an asset.

She is working on mapping the formal specification in domain concepts so that the reuser will see the translation of the formal specification in domain specific terms.

Participant 5

Participant 5 is performing a small test project in the reuse area. This project is being used as a test case for their reuse procedures. He has a domain analysis and engineering team that is building an Electronic Warfare and Intelligence domain architecture. After the architecture is in place, they will attempt to use it to build reusable components. The focus of their reuse technique is on the domain architecture rather than a library system.

Since they are in the early stages of reuse, there hasn't been much thought about testing and certification yet.

Participant 6

Participant 6 is currently creating a reuse process for his company. His application domain is cellular applications and his company is currently at an SEI level 2.

He is looking at code, design, architecture, and requirements reuse. A library system will be put into place to store assets.

Since he is in the early stages of design, all of this is yet to be completed.

He wants to include certification information along with the assets in the library. He has defined an elaborate process to certify an asset. An asset will undergo four steps of inspection prior to certification. All four steps must be successfully completed before an asset is considered certified.

1) The asset will be tested against its specifications,

2) The asset will be analyzed to ensure that it meets company standards,

3) The asset will undergo Fagan Inspection; a static testing method used to verify that the asset meets its requirements [FAG86],

4) The asset will be analyzed to ensure that it follows standard programming procedures.

When adding an asset to the library, it will go through a qualification phase, cost benefit analysis, and a certification phase to verify that it meets the criteria of the repository.

In his corporation, reusers have a monetary incentive for reusing assets. The tracking of this information is rather complicated. When reusers check out an asset from the library system, the life of that asset is tracked. When the system is delivered to the customer, the software is checked for verification of reuse. The reuser is then rewarded.

Certification is viewed as an incentive for reuse. He feels that assets with higher levels of certification are more likely to be reused.

[FAG86] Michael E. Fagan, "Advances in Software Inspections," *IEEE Transactions on Software Engineering*, Vol. SE-12, No. 7, July 1986.

Participant 7

This participant works with several domains: signal processing and programming environments. She is using reuse in the realm of a new program, but the reuse is limited to software tools (i.e. sorting utilities), I/O, mathematical calculations, etc. The entire application is 200,000+ lines of code. Three to four dozen Ada packages have been developed in house and are available for reuse. Approximately 20% of the tools that are being developed are used by more than one person.

Her company has no guidelines or incentive for reuse. Anything that a user can get "for free" is viewed as helpful to the software development effort.

This participant is trying to set up a reuse effort with little internal support. She is trying to set up a reuse repository to store the assets along with test information. Unfortunately, test information is not always available to her and she is not always able to incorporate it into the repository.

She has never thought about whether or not certification has a place in software reuse.

Participant 8

This participant is working within a single, specific, domain: Secure Operating Systems.

Her library contains design and other documentation on the asset, the asset's source, test code, and test result information. A configuration management process documents any asset modifications, modification rationale, review state, and current state of the asset.

During the rebuilding of their system, all tests are re-executed to uncover latent bugs.

External assets are rarely, if ever, used due to possible royalty issues. Also, the lack of design documentation, test information, or certification information for the asset is a deterrent. If there were a way of ascertaining the quality of external assets she might consider their use as she is inclined to not re-invent the wheel during development.

In the context of her application, certification is not a reuse issue because she "reuses" code developed for previous software builds on subsequent builds.

Participant 9

Participant 9 is developing test equipment in a corporation that is trying to reach SEI level 3. He didn't know their current SEI level, but presumed that it was 1.

For their reuse effort, they have a group of people that are surfing the Internet, looking for assets that can be reused. They have contacted both the Army and DOD about their reuse repositories. This group is looking for assets that have sufficient documentation and test plans and that can be easily reused. If an asset doesn't include documentation and testing information, the group will make a judgment call as to the suitability of the asset for reuse. If it is suitable, documentation and test plans will be created. The group is hoping that anything contained within a reuse library is sufficiently tested and documented so that a level of confidence can be achieved. If the asset isn't sufficiently tested, they'll test it themselves.

He is using Rational Rose for design and then storing the model information along with the asset and any information pertaining to the asset in Apex. Testmate is used to analyze the coverage of their test procedures. Additional certification tools would be used if they would decrease certification time and cost. When asked what he would look for in certification tools and he responded that he thinks any certification tool should interface well with Testmate and provide complementary capabilities.

On the issue of certification, he said that he would not use a non-certified asset. His reusers would rather use in-house assets than outside assets simply because the assets have been tested, have sufficient test documentation, and the reusers are already familiar with the asset.

Participant 10

Participant 10 is working in the domains of aerospace and process control.

For reuse he uses generic Ada packages that are under configuration management (CM). He doesn't perform what we would consider reuse because he is working on a single product. Reuse to him is reusing software previously developed and placed under configuration management. His application is highly focused and the only real modifications to software occur when the hardware has changed enough to require it. Most of his work is subcontracted out and it is up to the subcontractors to identify areas of reuse (if they use it at all).

He has no need for reusing outside software, and all internal software has already been sufficiently tested. Certification tools would not aid his development effort.

Participant 11

Participant 11 has, in place, a repository that contains flight, C3I, and general domains of software assets. The repository is set up using Internet, Mosaic, and the World Wide Web. The repository is filled with assets that have been developed in-house by in-house projects. It is the responsibility of each program to add reusable assets to the repository.

The repository contains data that describes each asset it contains. The descriptive examples are links to the asset's source, the name of the person that created the asset, and keywords for the asset.

As part of their engineering process, reusers look at the assets in the repository to identify potential reusable assets. To aid in keeping the reusers up to date on the state of the repository, a special internal group prints a department newsletter documenting the assets in the repository.

This same internal group is responsible for creating reusable flight and C3I assets. The assets are designed, if possible, to be reusable across both domains.

When asked about the benefits of certification tools and asset certification, the participant replied by stating that she thinks that, case by case, more mature assets would, over time, be noticed in the repository. Less mature assets would be "weeded out."

Participant 12

Participant 12 heads up a reuse effort that spans five divisions: surveillance, avionics, countermeasures, defense systems, and information systems. Her team consists of five people, plus herself. Each of the five are from one of the divisions.

This team meets weekly to discuss reuse issues across the company and its divisions. This reuse effort is currently in its infancy. Most of the items that she discussed with me will be implemented over the next year.

Rather than focus on reuse within each of the five domains, they are taking an opportunistic approach. They have drafted a reuse library guidebook that details the format for submitted assets.

A configuration board will be created to assess the assets that are submitted to the library and verify that the assets follow the reuse guidelines.

The reuse library is categorized by division and can be searched using keywords. Some effort has been spent defining the library's content, although not all of the specifics have been detailed. Some of the information within the library is: asset name, version/release, domain, keywords, abstract, dependencies, asset type, language, metrics (form fit functions), security type, *certification level,* and the division that created it. Unfortunately, the certification level information, although it will appear in the library, is one of the items that hasn't been defined yet.

When asked why a reuser would choose one asset over another, and she responded that the reuser would, most likely, choose an asset developed in their domain/division. If the assets returned from a library query had the same division, the reuser would probably focus on the abstract to determine which asset most closely matches the reusers requirements.

Personally, she thinks that certification would help in reuse but she doesn't know how it would work in "real life."

Participant 13

This participant believes that certification is the key to reuse. He is currently doing a feasibility experiment using four to five reusable libraries of approximately 10,000 lines of code (very approximate). He has not reached a "real world" stage for his ideas.

Assets that possess a predefined set of domain driven properties are considered certified. These properties identify such information as how an asset handles dynamic memory. If an asset does not meet these properties, it is not certified.

His experiment is within the domains of safety critical systems, embedded control systems, nuclear applications, and medical applications.

Along the lines of certification, he had several responses:

- Certification is an absolutely essential element of reuse

- If you can't trust it, you won't reuse it
- Although certification can be expensive, it is the key, missing part to reuse.

Participant 14

Participant 14 works in the shuttle design, trajectory analysis, and weight estimation domains. He is currently developing assets that are portable from one application to another.

At this stage, reusers are responsible for finding reusable assets, but his organization is starting to develop a reusable library. Currently they are using the Cosmic repository as an asset library.

Documentation and test cases are maintained for their internal software. The documentation also keeps track of asset revisions.

Unfortunately, he didn't know how certification would or could be used to benefit reuse.

Participant 15

This participant works in a small reuse group. The domains that they work in are electronic warfare, real-time embedded Ada systems, and command and control.

She is a member of a software engineering process group that is currently looking at asset information and design during the asset development and analyzing the reuse potential of the individual assets.

Their method of reuse is informal. Previous projects are searched for possible reusable assets. Project history data is used to aid in asset selection. Reuse is looked at as a way to reduce work and aid projects in meeting their schedules. There is no process or incentive for reusers to seek reusable assets. For current projects, they are attempting to build in reuse.

The engineering process group is developing a library containing the asset's code specification (Ada) as well as its functional specification. Any of the asset's limitations or problems will go into the header of the asset's code.

While she believes that certification information would be beneficial to reuse, she also thinks that past test history would aid in reuse.

Participant 16

Participant 16 is working with a very focused alarm monitoring application. Because of her highly focused application domain, she only reuses assets from previous application builds.

The library system that she uses has limited search capabilities and the asset information is limited to a brief description of the asset along with the assets header. Reusers search for assets using keywords.

No quality or test information pertaining to the asset is stored in the library.

When asked about certification, she responded favorably stating that it would be a good idea and that she could see how it would be useful. But, until this conversation, she had never thought about it.

Participant 17

Participant 17 builds stand-alone products in the missile and missile launcher domains. Currently they are at SEI level 1 and are working on obtaining a level 2 rating. Much of the reuse information he discussed is currently being defined and implemented.

He is performing domain analysis and building a reuse framework for his company.

He is examining ways to develop naturally reusable assets. These assets will be placed within a cataloguing system and will be identified by their requirements.

He is seeking reuse tools that can sit atop the CM system and development process. This tool will be used to search for assets within the CM system given a set of requirements.

The information within the library system will be very detailed. The source code along with test code and requirements information will be contained within the library. Assets that have been checked out will be tracked. If they are used, this will be noted by the library system. If they are unused, this, too, will be noted.

Much of an asset's information will be tracked, documented, and maintained after its inclusion in the reuse library: information ranging from bug fixes to the domain(s) it has been successfully reused in.

Participant 17 stated that no asset would be used without some level of certification. Without a test suite, documentation detailing the asset's errors and subsequent fixes, and without a requirements specification for the asset, he would not reuse it. All of this information would have to be created for assets not having it. It would be less effort to build the asset from scratch.

Participant 18

This participant is in the process of setting up a reuse repository in his company. In addition to software assets, the repository will include frameworks, documents, and tools. The application domains of the assets that will be stored in the repository are telecommunication and business.

Participant 18 thinks that robustness (defined to be mean time between failure), reliability, and accurate documentation are key for software reuse.

His company has three levels of certification. Software is given a level according to its ranking in five areas of concern: process conformance, functional completeness, functional availability, architectural conformance, and organizational commitment. He views certification as a means to record an asset's life cycle rather as a means of keeping assets out of a repository.

No special testing or analysis would be done in addition to whatever had occurred during development of the asset. He said that certification would be handled on a case by case basis and that assets that didn't meet one of the three certification levels would still be included in the repository. He felt that ultimate judgment should be left to the user.

Miscellaneous Participants

The remaining participants fall under one of four categories: not currently reusing (due to funding cuts or reuse is not being pushed by the company), never heard of reuse, simply did not want to talk to me, and unable to be contacted. There are fifteen of these participants.

The two people that didn't want to discuss reuse left me with the definite feeling that they didn't want to divulge the fact that they weren't doing reuse.

Two people gave me some quotable material:

> "Reuse? We're not doing anything by that name."
>
> "Not really." "Uh, not really means, really, no."

The rest of the participants were either unable to be contacted or were not currently active in reuse.

Appendix D - State-of-the-Art Report on Reuse Libraries

State-of-the-Art Report on Reuse Libraries

Many reuse programs were initiated in the late 80's as a potential solution to the spiraling upward costs of producing custom software solutions for the Department of Defense. The Army sponsored a library called RAPID (Reusable Ada Products for Information Systems Development) which is now known as the Army Reuse Center. Over the last few years the processes and assets of these libraries have evolved due to changing missions and goals of their sponsors as well as introduction of new technologies. In addition, the widespread availability of the Internet has encouraged the establishment of reuse libraries as marketplaces for commercial vendors and created a much larger and disparate pool of potential reusers.

Other projects related to reuse were sponsored by the Defense Advanced Research Project Agency (DARPA). The goals of STARS program were to develop repeatable architecture-based reuse processes, encourage development of domain analysis methods and serve as a catalyst for technology transfer. The STARS program produced a handbook The Conceptual Framework for Reuse Processes to serve as a guide for developing reuse capabilities within an organization. They sponsored development and application of four domain analysis methods:

- Feature Oriented Domain Analysis at Carnegie Mellon's Software Engineering Institute.
- Synthesis at the Software Productivity Consortium
- Organization Domain Modeling at Organon Motives
- Domain Analysis Process Model by Ruben Prieto-Diaz

STARS also sponsored a Technology Transition Affiliates program which transferred their knowledge to many of the government and commercial organizations now running reuse libraries (ASSET, CARDS, DISA).

The STARS program also funded the secretariat activities of the Reuse Library Interoperability Group (RIG). The RIG is a working group of commercial and government representatives dedicated to producing standards which allow reuse libraries to interoperate. The working group currently has 3 standards forwarded to IEEE for ballot. These are:

P1420.2	Basic Interoperability for Data Models for Reuse Libraries
P1420.1	Asset Certification Framework

The Basic Interoperability Data Model (BIDM) defines the minimal set of information about assets that reuse libraries should be able to exchange to support interoperability. The BIDM (Figure 1) is intended as a meta-model which defines attributes and relationships for any kind of reusable asset. It does not contain information about a reuse library's data model or communication protocol. The Asset Certification Framework is an extension to BIDM which models information specific to various methods of certification.

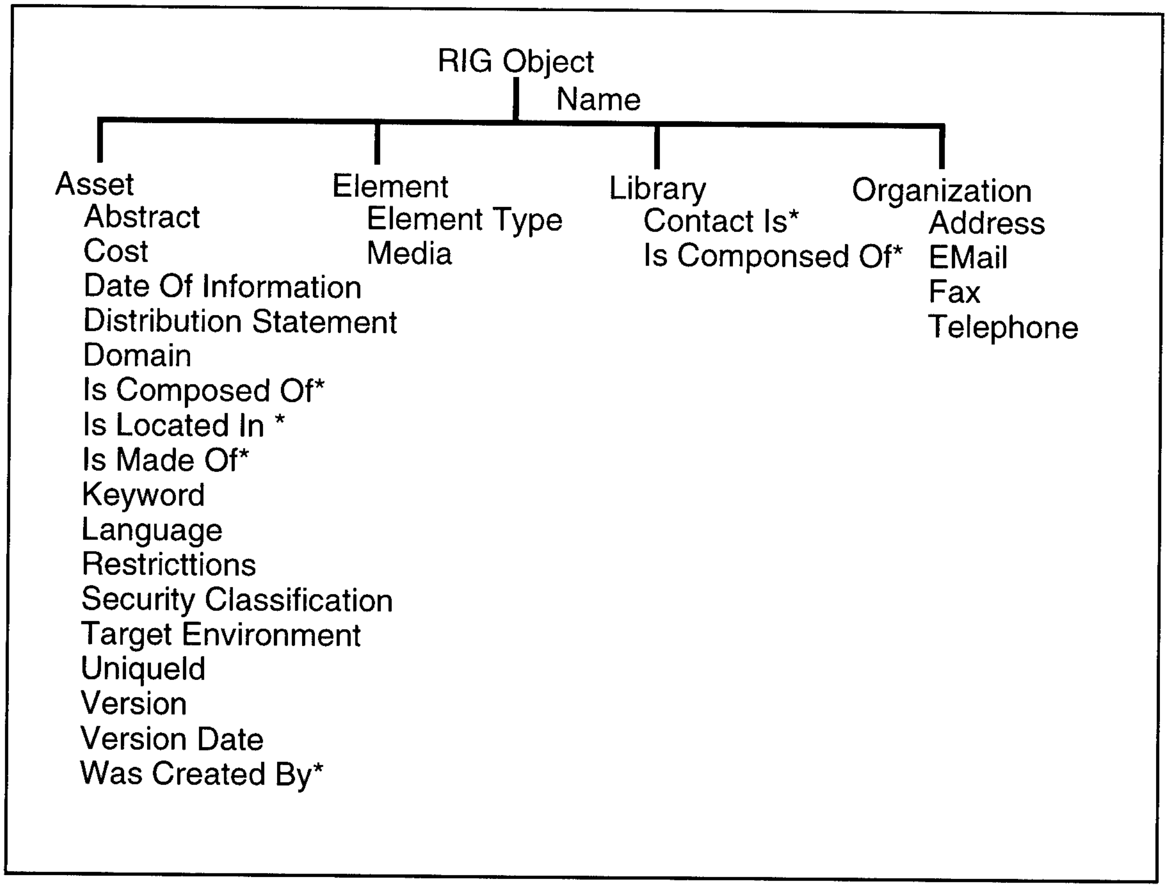


Figure 1. The BIDM meta model

This report is intended as an introduction into the current practices of both commercial and government reuse libraries. It examines:

- Business model - What is the organization's goals and mission? Who are the target customers
- Asset Production - Who produces and donates assets? What standards or specifications must be met?
- Asset Selection - What tools and selection methodology are used?
- Asset Certification - What methods, techniques, tools or criteria are used?

The following libraries were surveyed:

IBM

Raytheon

Netlib

ASSET

Army Reuse Center

ELSA/MORE

DSRS

IBM's Component Store

Business Model - IBM's largest reuse project is to provide The Reusable Software Component Store on the Internet to support external vendors supplying complementary parts to IBM Object Technology products. While the library doesn't support one particular domain, the parts can generally be categorized as OLE, ParcPlace and OpenDoc components as well as Visual Basic class libraries and frameworks. IBM, in conjunction with CyberSource Corporation, is sponsoring this library by paying for its creation, operation and maintenance. IBM's motivating business case for sponsorship is to stimulate use of their COTs products by providing a venue for their customers to obtain additional parts of applications rather than develop these parts again. An example is a dictionary or thesaurus part for a VisualAge application. Customers which use these parts can then market their own developed parts in this marketplace. Since such parts are small and low cost, it would not be cost effective for vendors to individually market and distribute them.

Asset Production - Parts are produced by vendors external to the IBM organization.

Asset Selection - The parts will be purchased and electronically delivered on the Internet. IBM has chosen to outsource the library creation, operation and maintenance to an organization called *Softwarenet.* The http site is http://components.software.net. *Softwarenet* was recently identified by the Wall Street Journal as the largest provider of Software on the Internet. Their large corporate clients include not only HP and IBM but Microsoft is considering them as a potential pilot site. For IBM's component store the selection tool is a user interface front end to a relational database. They are using a faceted classification scheme with the following facets:

Function Area

Vendor

Language

Visual Tools

Platforms

National Language

Architecture

Asset Certification - The vendor submits the software (Figure 2) and accompanying information template (Figure 3) The file(s) are scanned for viruses using Norton's Utilities and then compressed and encrypted for delivery on the Internet. No quality checks are done. IBM plans to collect user feedback to remove undesirable assets. Standard commercial software agreements limit IBM's liability in situations of non-performing software parts.

Technical specifications for electronic delivery:

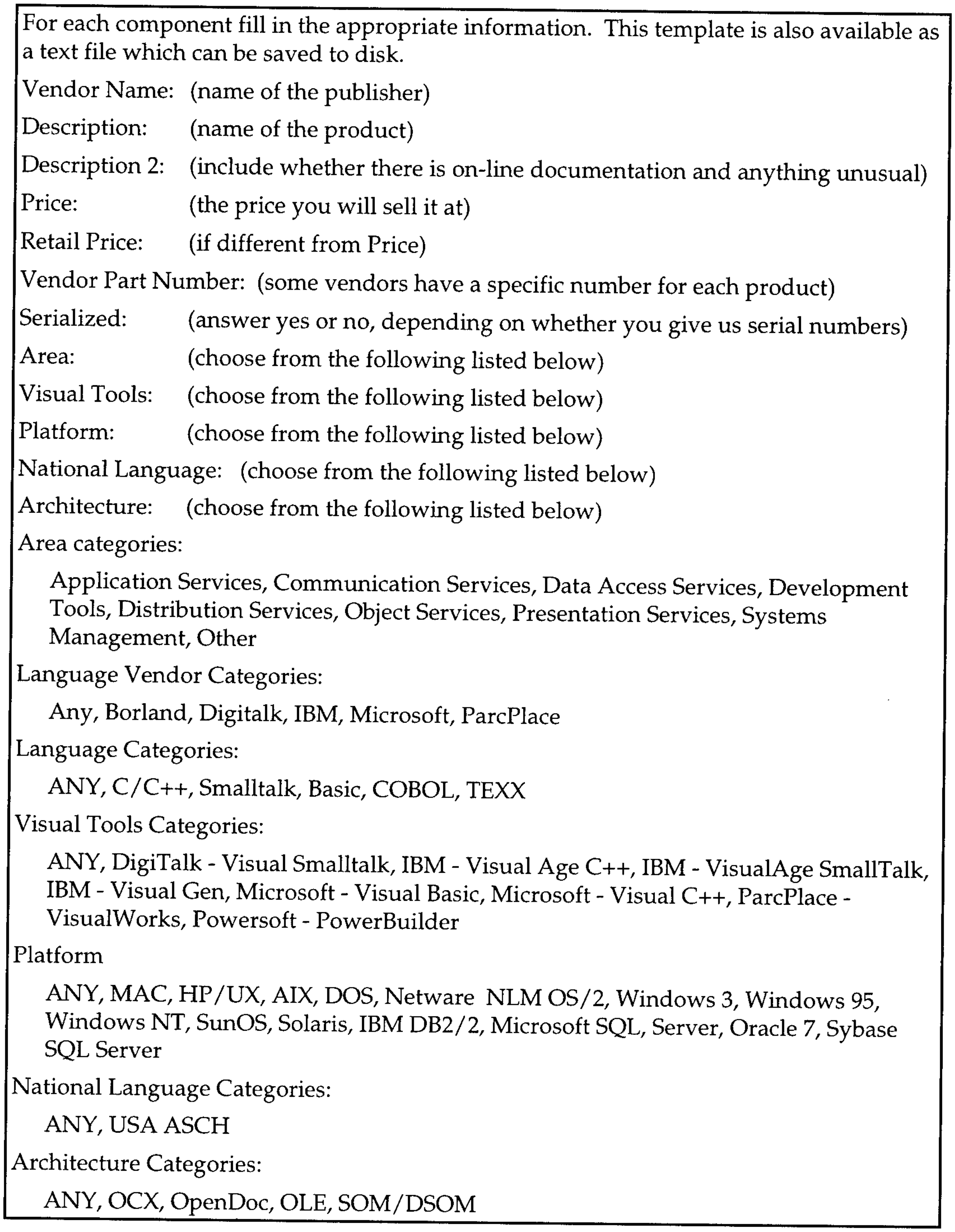
1. Complete the component description template for each component. This template can be found at the end of this page.
2. Create a document which includes your company name address, phone number, and a contact person and his phone and/or email address.
3. Zip all relevant files (the template(s), the component(s), license.txt, any documentation and any images into one file.
4. UUEncode the file and send it to skp@software.net. Alternatively, you may send the zipped file as a MIME attachment to the same address. If you have an FTP site, we can also get the zipped file from your site. Send mail to skp@software.net with the location and password of the site.

Technical specifications for diskette delivery:

1. Create a directory for each component
2. In each directory, complete the component description template for each component. This template can be found at the bottom of this page.
3. Create a subdirectory within each component directory called "data" and put all component data files into the data directory.
4. Into the main directory place all other relevant materials. This includes the license.txt, any documentation, and any images.
5. Create a document which includes your company, name, address, phone number, and a contact person and his phone and/or email address.
6. Send the diskette(s) to:

CyberSource Corp.
Attention: Reusable Software Component Market Manager
1050 Chestnut Street, Suite 200
Menlo Park, CA 94025

Figure 2. Technical Specifications for Submission of Software Products



For each component fill in the appropriate information. This template is also available as a text file which can be saved to disk.

Vendor Name: (name of the publisher)

Description: (name of the product)

Description 2: (include whether there is on-line documentation and anything unusual)

Price: (the price you will sell it at)

Retail Price: (if different from Price)

Vendor Part Number: (some vendors have a specific number for each product)

Serialized: (answer yes or no, depending on whether you give us serial numbers)

Area: (choose from the following listed below)

Visual Tools: (choose from the following listed below)

Platform: (choose from the following listed below)

National Language: (choose from the following listed below)

Architecture: (choose from the following listed below)

Area categories:

Application Services, Communication Services, Data Access Services, Development Tools, Distribution Services, Object Services, Presentation Services, Systems Management, Other

Language Vendor Categories:

Any, Borland, Digitalk, IBM, Microsoft, ParcPlace

Language Categories:

ANY, C/C++, Smalltalk, Basic, COBOL, TEXX

Visual Tools Categories:

ANY, DigiTalk - Visual Smalltalk, IBM - Visual Age C++, IBM - VisualAge SmallTalk, IBM - Visual Gen, Microsoft - Visual Basic, Microsoft - Visual C++, ParcPlace - VisualWorks, Powersoft - PowerBuilder

Platform

ANY, MAC, HP/UX, AIX, DOS, Netware NLM OS/2, Windows 3, Windows 95, Windows NT, SunOS, Solaris, IBM DB2/2, Microsoft SQL, Server, Oracle 7, Sybase SQL Server

National Language Categories:

ANY, USA ASCH

Architecture Categories:

ANY, OCX, OpenDoc, OLE, SOM/DSOM

Figure 3. Component Description Template

Raytheon

Business Model - Raytheon has an internal, non-commercial reuse library with assets to support its government projects. Although the assets encompass a broad range of domains, most of them are related to C4I which is key business are for Raytheon. The assets are no only software; they include products from all phases of the lifecycle, particularly software development plans. There are also proposals and COTs and GOTs product evaluation. The business objective is that Raytheon's government projects in these domains will be able to reuse these assets to lower their cost and compress their schedules.

Asset Production - Most assets are the result of internal donations from company projects. The library team supplements these with COTs products and shareware assets from the Internet.

Asset Selection - Raytheon's library uses the World Wide Web with a firewall which only allows access to its employees. All assets are described in a relational database using the BIDM-meta-data. The search engine tool is Topic by Verity and Adobe Acrobat is used to display documents. Raytheon considers the design and layout of its search and meta-data screens to be company sensitive information and does not release copies of them to the public in order to maintain a competitive edge. Other display and translator tools are used as needed.

Asset Certification - No certification process is applied to COTs software assets. Shareware assets found on the Internet are compiled and executed by the library team. All internal software donation assets are subjected to the 21267A inspection process by the project team before they are donated. The library team does not currently certify these assets; however, they are considering implementing a certification program similar to ASSET's.

Netlib

Business Model - University of Tennessee's WWW digital library consists of multiple libraries and supports two separate projects. Netlib was established in 1985 and is managed by the University of Tennessee and AT&T Bell Labs. Its primary purpose was to serve as a repository of mathematical routines. University of Tennessee is also a member of the Center for Research on Parallel Computation (CRPC) and in 1994 became one of virtual distributed web sites for the National HPCC Software Exchange (NHSE). The NHSE is a distributed collection of software, documents, data and information of interest to the high performance computing community. The NHSE has the goals of facilitating the development of discipline-oriented software and document repositories and promoting contributions and use via the World Wide Web.

Important Note: Netlib is a very successful reuse library. As of November 1995, it has had 7.8 million accesses to its library. (This includes browses of the http page.)

Asset Production - Components for Netlib are donated primarily by the members of the academic mathematical and engineering research community. There are approximately 147 separate libraries Table 1.

Table 1. Netlib Libraries

Netlib Libraries

a
access
aicm
alliant
amos
ampl
anl-reports
apollo
benchmark
bib
bibnet
bihar
blacs
blas
bmp
c
c++
cephes
chammp
cheney-kincaid
fitpack
floppy
fmm
fn
fortran
fortran-m
fp
gcv
gmat
gnu
go
graphics
harwell
hence
hompack
hpf
hypercube
ieeecss
ijsa
image
intercom
mds
microscope
minpack
misc
mpfun
mpi
na-digest
napack
netlib
news
numeralgo
ode
odepack
odrpack
opt
p4
paragraph
paranoia
parkbench
parmacs
pascal
random
research
scalapack
sched
scilib
seispack
sequent
sfmm
shpcc9
slap
slatec
sodepack
sparse
sparse-blas
sparspak
specfun
spin
srwn
stoeplitz
stringsearch
svdpack

Netlib Libraries

clapack
commercial
confdb
conformal
contin
crc
ddsv
dierckx
diffpack
domino
eispack
elefunt
env
f2c
fdlibm
fftpack
fishpack
itpack
jakef
kincaid-cheney
la-net
lanczos
lanz
lapack
laso
lawson-hanson
linalg
linpack
list
lp
machinesx
magic
maspar
pbwg
pdes
performance
photo
picl
pltmg
poly2
polyhedra
popi
port
posix
pppack
presto
problem-set
pvm3
quadpack
templates
tennessee
testbook
toeplitz
toms
typesetting
uncon
vanhuffel
vfftpack
vfnlib
voronoi
xmagic
xnetlib
y12m

Development of the NHSE repository is carried out by the federal High Performance Computing and Communications Program (HPCC). Components for NHSE are developed by programs sponsored by the HPCC. These programs include:

- Advanced Software Technology and Algorithms (ASTA)
- High Performance Computing Systems
- Information Infrastructure Technology and Applications (IITA)

The components are grouped under the domains of:

- Data Analysis and Visualization
- Parallel Processing Tools (Analyzing Environments, Languages, Compilers, Libraries, Runtime Systems and Source Code)
- Scientific and Engineering Routines
- Numerical Programs and Routines (Computational Geometry, Linear Algebra, Optimization, Partial Differential Equations

The components are grouped under the domains of:

- Data Analysis and Visualization
- Parallel Processing Tools (Analyzing Environments, Languages, Compilers, Libraries, Runtime Systems and Source Code)
- Scientific and Engineering routines
- Numerical Programs and Routines (Computational Geometry, Linear Algebra, Optimization, Partial Differential Equations)

Asset Selection - All assets are published on the World Wide Web and are accessible to anyone in the Internet community.

Asset Certification - Authors submitting software to the NHSE are required to fill out a Completed NHSE Software Submission Form (Figure 5) and request one of the following review levels:

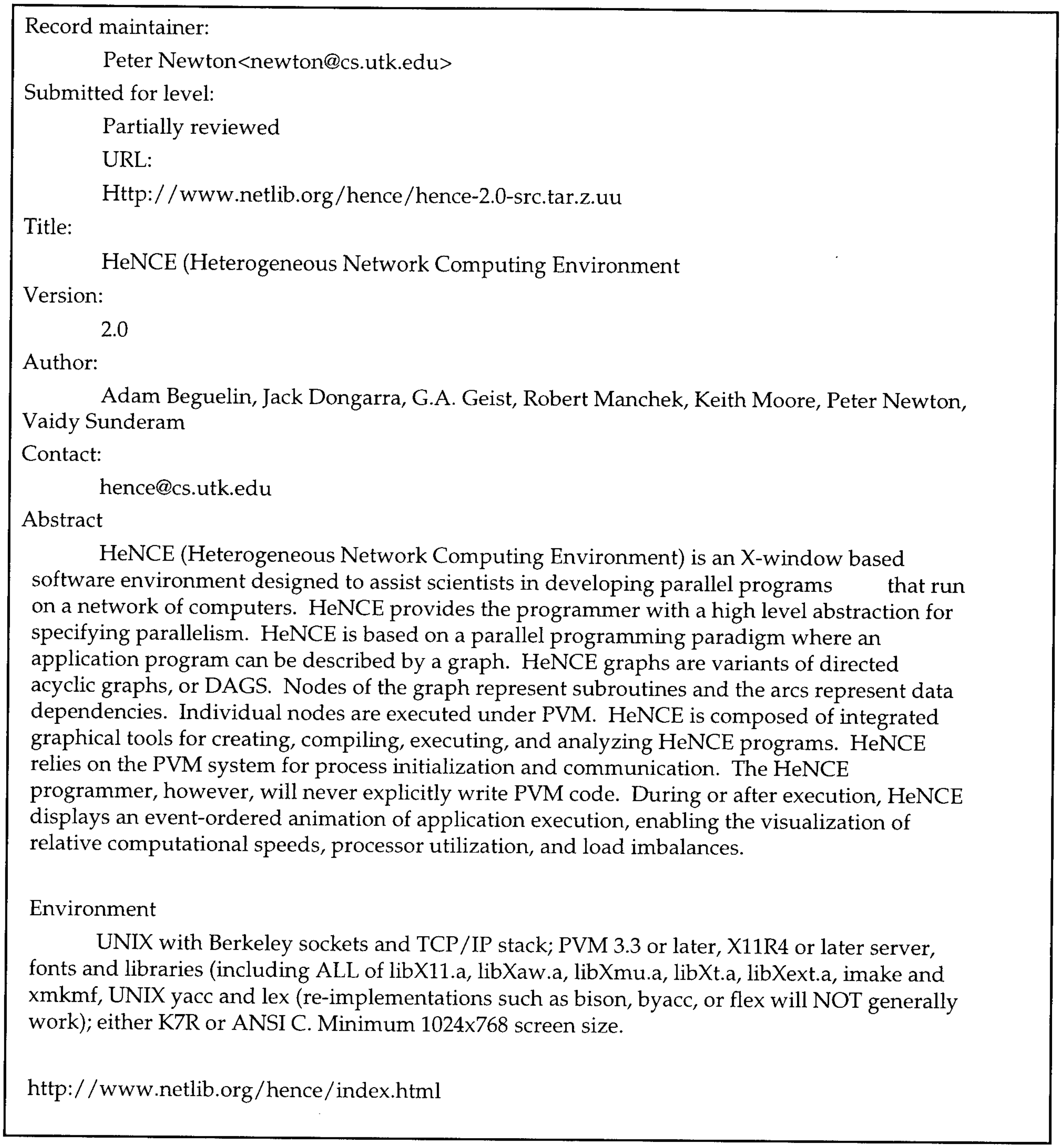
> Unreviewed - The submission has not been reviewed by the NHSE for conformance with software guidelines. This classification is for unreviewed software available on an "as is" basis.
>
> Partially reviewed - The submission has undergone a partial NHSE review to verify conformance with the scope, completeness, documentation, and construction guidelines. These particular guidelines are those that can be verified through a visual inspection of the submission.
>
> Reviewed. The submission has undergone a complete NHSE review to verify conformance with the full review criteria. This classification requires peer-review testing of the submitted software.

The Partially reviewed and Review levels require the use of PPG authentication mechanism if the software is sent automatically. (Use of PPG is also highly recommended for Unreviewed submissions). Software sent via surface mail requires the signature of a notary public.

To receive the Partially reviewed rating, software submitted to the NHSE should conform to the following guidelines:

> Scope: Software submitted to the NHSE should provide a new capability in numerical or high-performance computation or in support of those disciplines.
>
> Completeness: Submissions must include all routines and drivers necessary for users to run the software. Test problem sets and corresponding drivers must be included if the software is to undergo peer-review testing for the Reviewed level. Source code for widely available software used by the submission, blas and lapack for example, need not be included as part of the submission.
>
> Documentation: The software contains complete, understandable, correct documentation on its use.
>
> Construction: Submissions must adhere to good mathematical software programming practice and, where feasible, to language standards. Software should be constructed in a modular fashion to facilitate reusability. The use of language checking tools, such as pfort or ftnchek or recommended.

To be accorded the reviewed status, the software must first have been accorded the partially reviewed status. This precondition ensures that reviewers will be able to access all the information needed to carry out the review over the National Information Infrastructure.



Record maintainer:

Peter Newton<newton@cs.utk.edu>

Submitted for level:

Partially reviewed

URL:

Http://www.netlib.org/hence/hence-2.0-src.tar.z.uu

Title:

HeNCE (Heterogeneous Network Computing Environment

Version:

2.0

Author:

Adam Beguelin, Jack Dongarra, G.A. Geist, Robert Manchek, Keith Moore, Peter Newton, Vaidy Sunderam

Contact:

hence@cs.utk.edu

Abstract

HeNCE (Heterogeneous Network Computing Environment) is an X-window based software environment designed to assist scientists in developing parallel programs that run on a network of computers. HeNCE provides the programmer with a high level abstraction for specifying parallelism. HeNCE is based on a parallel programming paradigm where an application program can be described by a graph. HeNCE graphs are variants of directed acyclic graphs, or DAGS. Nodes of the graph represent subroutines and the arcs represent data dependencies. Individual nodes are executed under PVM. HeNCE is composed of integrated graphical tools for creating, compiling, executing, and analyzing HeNCE programs. HeNCE relies on the PVM system for process initialization and communication. The HeNCE programmer, however, will never explicitly write PVM code. During or after execution, HeNCE displays an event-ordered animation of application execution, enabling the visualization of relative computational speeds, processor utilization, and load imbalances.

Environment

UNIX with Berkeley sockets and TCP/IP stack; PVM 3.3 or later, X11R4 or later server, fonts and libraries (including ALL of libX11.a, libXaw.a, libXmu.a, libXt.a, libXext.a, imake and xmkmf, UNIX yacc and lex (re-implementations such as bison, byacc, or flex will NOT generally work); either K7R or ANSI C. Minimum 1024x768 screen size.

http://www.netlib.org/hence/index.html

Figure 5. Example of a Completed NHSE Software Submission Form

The author of partially reviewed software may submit his software for full review by filling out and submitting the NHSE submission change form and selection the Reviewed level. The software will be reviewed according to the following criteria:

> Documentation: The software contains complete, understandable, correct documentation on its use.
>
> Correctness: The software is relatively bug-free and works as advertised on all provided data sets and on data sets provided by the reviewer according to the documentation.
>
> Soundness: The methods employed by the software are sound for solving the problem it is designed for, as described in the documentation.
>
> Usability: The software has an understandable user interface and is easy to use at the level of a typical NHSE client.
>
> Efficiency: The software runs fast enough, in that slow speed does not make it an ineffective tool.

Once software is submitted, it will be assigned by a semi-automatic procedure to an associate editor (one of the editorial board members). That editor will recruit two to six reviewers to review the software according to the criteria from the last section. When the review(s) are returned, the associate editor will make the final decision about whether to accept the software and will inform the author of the decision. If the software is accepted, the assigned associated editor will prepare the review abstract for use by the NHSE.

After the software is reviewed, one of two things happens. If it is not accepted, the author will be so informed and anonymous copies of the review or reviews will be provided. If it is accepted, the author will be shown a review abstract summarizing the reviewer comments. This abstract will be available to anyone who accesses the software through the NHSE. If the author finds the abstract unacceptable, he or she may withdraw the software and resubmit it for review at a later date.

SAIC/ASSET

Business Model - SAIC/ASSET (Asset Source for Software Engineering Technology) was originally established under the LSTARS Program to develop a national software repository for reusable software assets and information about software reuse. ASSET initially specialized in reusable Ada Software components., Ada standards and bindings, and in documents written specifically to promote software reuse or documents describing reuse products, processes, procedures or lessons learned. ASSET is now transitioning to a private enterprise as a division of SAIC. They offer the following services via the World Wide Web:

-Worldwide Software Resource Discovery (WSRD). The WSRD library contains documents, software components, vendor advertisements and information on software reuse practices. There is information on both public domain and commercial assets. All software components in the WSRD are free but a user must have an ASSET account to download them.

-On-Line Information of Reusable Technology, Publications and Conferences

-Reusable software Asset Brokerage Services is ASSET's new commercial Electronic Market Place which provides a forum for vendors to sell products or set up a WWW home page advertisement.

Asset Production -Assets are produced by organizations or individuals external to the organization. For the WSRD most assets are donated by government programs and are free of charge. ASSET's criteria for evaluating assets is outlined in the document "Criteria and Implementation Procedures for Evaluation of Reusable Software Engineering Assets" prepared by Software Engineering Technology, Inc. and dated March 28, 1992. Producers are expected to provide documentation (Figure 6) and meta-data (Figure 7), based on the BIDM, about the asset. Most assets are either software components or documents.

an abstract

a user's guide or instructions on how to use

a list of files making up the asset (preferably in compilation order)

installation/implementation instructions

sample input/output

design and/or requirements documents

test programs, procedures and/or results

description of the environment under which asset was developed/tested

known limitations of software

a list of tools needed to interpret or use the asset

warranties or disclaimers

statement of distribution rights/licenses

list of special formats of files (e.g. PostScript, InterLeaf, SGML)

Figure 6. Recommended Documentation

```
name =>
alternate_name =>
version =>
release_date =>
asset_size =>
asset_type =>
distribution_code =>
domain =>
keyword =>
referencenumber=>
computer_language=>
national_language =>
computer_environment
=>
format=>
author name=>
abstract=>
support_available=>
```

```
contact_person=>
begin person
  name=>
  street_address=>
  city=>
  state_code=>
  zip_code
 country
  phone
  fax
  email
end person
producer=>
  begin organization=>
  name
  streetaddress=>
  city=>
  state_code =>
  zip_code =>
  country =>
  phone =>
  fax =>
  email =>
end organization
```

```
supplier =>
begin person
  name =>
  street_address =>
  city =>
  state_code =>
  zip_code =>
  country =>
  phone =>
  fax =>
  email =>
  end person
  supply_date =>
end person
```

Figure 7. Required Meta-Data

Products submitted to the Electronic Marketplace require company and product logos and abstracts as well as the following meta-data per product offering:

- Version number
- Keywords
- Computer Languages

- Hardware/operating system/compiler
- Native Language (available documentation languages)
- Product Release Date
- URL
- E-mail address

Asset Selection - Selection is done through keyword search. The search is done all words in the title, abstract, meta-data as well as the asset itself. The library screens also offer four upper level screen choices for selection under Domain, Collection, Asset Name, and Asset Identifier.

Asset Certification - Under advice from their lawyer, ASSET do not call their levels "certification" levels but rather "evaluation" levels. The term certification implies a legal liability that evaluation does not. Level 1 evaluation checks that the meta-data, documentation and asset exist and are complete. Level 2 checks that the meta-data and documentation actually match the asset. Level 3 provides assurance of the performance of the asset.

The Army Reuse Center

Business Model - The Army Reuse Center (ARC), formerly a division of the Software Development Center Washington (SDC-W) and originally named the Reusable Ada Products for Information Systems Development (RAPID), was established as a directorate of the US Army Information Systems Software Center (USAISSC) on 1 January 1994. Its stated mission is "to develop, implement, maintain, and administer a total reuse program supporting the entire software development life cycle, and thus help the Army fulfill the Department of Defense (DoD) objective to institute reusable, maintainable, and reliable software."

Although ARC's origins are as a dialup reuse library to support reuse and development of Ada software assets, it is now located on the WWW (http://arc_www.belvoir.army.mil) and provides a full range of products and services. These include:

- Reuse Management

- Reuse Education
- Domain Analysis
- Domain Implementation
- The ARC Library

The services are offered for a fee to cover the ARC's costs. For example, the startup costs to provide domain repository services are 25K a year with recurring annual costs of $20K a year.

Asset Production - Assets in the ARC library include submissions from supporting DoD reuse programs and commercial reusable assets. They include requirements, specifications, architectures, design diagrams, code and documentation. The Reuse Implementation Strategy for Horizontal Reuse across the US Army's Vertical Domains has targeted the following organizations for development of reuse programs to donate assets:

- PEO Aviation
- PEO Armored systems Modernization
- PEO Command and Control Systems
- PEO Communications
- PEO Cruise Missiles and Unmanned Aerial Vehicles
- PEO Field Artillery Systems
- PEO Missile Defense
- PEO Standard Army Management Information System
- PEO Tactical Missiles
- PEO Tactical Wheeled Vehicles
- Simulation, Training and Instrumentation Command

Highly successful reuse programs within PEO STAMIS and the USAISSC have donated many of the assets.

Asset Selection - The ARC Library uses DISA's Defense Software Repository System (DSRS) as their library and selection tool. The search mechanism is a faceted classification scheme in which the user interface runs on top of an RDBMS. (See DSRS description for a list of facets and more complete description of the tool.) The user may search on metrics such as reusability, maintainability, portability, number of uses, problem reports, and level of certification.

Asset Certification - The following activities are performed by Army Reuse Center Personnel for software assets in the ARC library at these certification levels:

- Level 1 - Code is installed "as is"

  - Code does not compile

  - Certification forms are complete.

  - An abstract is written.

  - AdaMat is run.

  - Cataloging (facet terms) is finished.

  - In-line comments are added for clarity.

- Level 2 - Level 1 plus code compiles correctly.

- Level 3 - Levels 1 & 2 plus test drivers are provided.

- Level 4 - Levels 1, 2 & 3 plus documentation such as reference and users' manuals are provided.

The Army Reuse Center is planning on adding additional quality metrics to these levels in FY 96.

ELSA/MORE

Business Model - The origins of ELSA/MORE can be found in AdaNET established as a direct reply to an unsolicited white paper to congress, proposing the creation of a model clearinghouse to answer the 1987 Congressional Ada Mandate. As originally envisioned, the clearinghouse would provide users of the Ada language with information and repository services while contributing to NASA's own use and adaptation of Ada software. AdaNET would therefore, enhance NASA's software development efforts and eventually transfer software technology to the public domain. AdaNET's mission was simply to house Ada code that could be extracted and incorporated as building blocks into development of new systems. AdaNET was part of the Repository Based Software Engineering (RBSE) program, sponsored by the Technology Utilization Division at NASA Headquarters and administered by the Johnson Space Center. In addition to operating a software lifecycle repository, RBSE promotes software engineering technology transfer, academic and instructional support for reuse programs, the use the common software engineering standards and practices, software reuse technology research, and interoperability between reuse libraries/repositories.

As the project matured, the processes, products, mechanisms, and tools developed to support the early vision became as important, if not more so, than the component holdings themselves. Thus the original vision has been adjusted from a solely software distribution repository to a process driven library, supporting various reuse methodologies and models. The program evolved into a software engineering environment which supports reuse, engineering environment which supports reuse, reengineering, and domain analysis. As a result of this evolution, the name AdaNET no longer adequately represented the project's scope. To better align the project and the library with the technological advances of the Internet eta, AdaNET was retitled Electronic Library Services and Applications, or ELSA. ELSA is a service, also NASA funded, that is located on the WWW (http://rbse.mountain.net) provided by MountainNet, a commercial contractor, which provides access to a large selection of software. MountainNet is charged with commercializing ELSA library mechanism MORE into a product called MOREplus.

In addition to commercializing MOREplus, MountainNet has transitioned this technology by using it to set up nine additional reuse libraries. There are three at commercial companies (bank, aerospace, and computer manufacturer) two DoD, and four additional NASA sites. MountainNet also sells commercial software through their Pinnacle Mall WWW site.

Asset Production - Assets in ELSA have been produced by NASA programs or are Internet shareware identified by MountainNet staff. Assets from the Internet are not physically stored in the ELSA library. Instead only meta-data about these assets and URL pointers to their location on the Internet are stored. There is currently no way to verify when these assets at other locations have been moved or changed. ELSA/MORE is also interoperable with ASSET's library. Again the assets are not physically stored in the library, rather the library mechanism provides the way to view distributed assets at a more remote location. Figure 8 shows various classes and collections of the ELSA/MORE library.

MORE Class Browser

Sub-Classes:

- Articles and Newsletters
- Bibliographies
- Books and Proceedings
- Catalogs
- COTS Toos and Environments
- Directories
- External Resources
- External Software
- External Tools
- Reports and Papers
- Software
- Standards
- Terminology
- Tutorials
- User Guides

MORE Collections (alphabetic)

Browsers:

- Customer Support
- Database Gateways
- HTML Tools
- Legal Issues
- Libraries and Servers
- Library Interfaces and Protocols
- Main Collection
- Math
- Matrices
- Mnet Tutorials
- Polynomials
- Reuse
- Routines and Algorithms
- Search
- Search Engines
- Server Software
- Sort
- Statistical Packages
- Viewers
- WWW Information and Utilization

Figure 8. Classes and collections of the ELSA/MORE library

Asset Selection - The ELSA library mechanism is a product called MOREplus. MOREplus is an information management tool consisting of a set of user interface executables that operate in conjunction with hypertext servers to provide access to a relational database. Access to data is achieved through hypertext links that take the user to database records or to sites maintained on the Web for information not physically sorted within the database. Access to proprietary or sensitive data can be restricted to designated groups of users and administrators. MOREplus uses both natural language and pattern match searches as well as hierarchical and alphabetical classification browsers. The collection of assets that is open for the general public can be viewed by:

- MORE Collection Browser
- MORE Class Browser
- MORE Collections (alphabetic)
- MORE Collections (hierarchical)
- MORE Natural Language Search
- MORE Pattern Search

Asset Certification - The ELSA repository uses a process called AQUIP (Acquisition, Qualification, Utilization, and Investigation Process) to qualify non-commercial assets which are actually stored in the repository (versus those at distributed sites). Phase 1 involves classification by meta-data. Phase 2 checks for completeness. The software tools ADAL and FDADL are used on Ada and FORTRAN software respectively. Phase 3 tests the assets for functionality and interfaces and provides asset usage information.

DISA's Defense Software Repository System (DSRS)

Business Model - DISA's software reuse program is primarily focused on the enhancement of its tool, the DSRS, and supports the installation and maintenance of this tool at various DoD sites. The DSRS is an accredited automated repository for storing and retrieving reusable assets. Reuse programs using this tool are:

- Naval Undersea Warfare Center (NUWC) - Newport, RI
- Maxwell AFB - Montgomery, AL

- Ft. Sill, OK
- McClellan AFB, CA
- Defense Logistics Agency (DLA) - Columbus, OH
- National Security Agency (NSA) - Ft. Meade, MD
- NCTS-Washington - Washington, DC
- DISA Software Reuse Center (SRC)- Falls Church, VA
- Army Reuse Center - Fort Belvoir, VA
- US Marine Corps - Quantico, VA

DISA's repository is located on the WWW (http://ssed1.ims.disa/mil/srp/dsrspage.html). It serves as a central collection point for quality assets, and facilities software reuse by offering developers the opportunity to match their requirements with existing software products. Although primarily used to support reuse and development of object-oriented Ada development products, the DSRS can support storage and retrieval of other than Ada related products.

Asset Selection - Users describe their requirements, using the faceted classification scheme, through a series of menu-driven screens. The repository identifies one or more suitable software assets from its collection. The DSRS provides the user with the capability to browse an individual asset or analyze a group of assets. Users may view the abstract, classification description, supporting documentation, and numerical measures for each asset. Additionally, the DSRS provides an on-line help facility, asset relationship and dependence information, session maintenance, and user suggestion facility. The list of facets available to the user are:

- ALGORITHM (e.g., binary, conversion, linear, shell, etc.)
- CERTIFICATION LEVEL (e.g., level_1, level_2, support etc.)
- COMPONENT TYPE (e.g., data_file, design, implementation, etc.)
- DATA REPRESENTATION (e.g., alphabetic, binary, string, etc.)
- ENVIRONMENT (e.g., AT&T 3B2/UNIX,IBM PC VAX/VMS, etc.)

- FUNCTION (e.g., compile, sort, sum, truncate, update, etc.)
- LANGUAGE (e.g., Ada, Assembly,Booch, Cobol, Yourdon, etc.)
- OBJECT (e.g., accounting, array, binding, model, vector, etc.)
- UNIT TYPE (e.g., dbms, program, subsystem, types_package, etc.)
- ORIGINATOR (e.g., Army_Reuse_Center, Gunter-AFB, STARS, etc.)

The list of all facet terms for each facet is too lengthy to be provided here.

Asset Production - The DSRS staff does not produce assets and in the past assets have been submitted by various DoD projects. The program is now focusing on interoperation of reuse libraries to provide their library users with access to more assets through a virtual library system. ASSET, CARDS and DSRS reuse libraries have developed an interoperability capability as a first step toward realizing the DoD Software Reuse Initiative's goal of building a "virtual library" A user interacts with the virtual library through the local user interface. It is not necessary for the user to know where the asset resides, nor even to know that it is not stored locally. The ability of a DSRS user to extract a particular asset from a remote library is based on the distribution class (group) privileges the user has been assigned.

Asset Certification - DSRS adopted the same certification levels as the Army Reuse Center. There are possible plans to update or augment these levels in the near future.

Conclusion

What conclusions can be reached by this brief survey of reuse libraries? Certainly Internet technology has drastically changed the methods of operation of reuse libraries. Most libraries now reside on the WWW and use hypertext technology to publish their assets. Libraries are utilizing the distributive technologies of the Web to provide a wider variety of assets to their user. Most libraries rely on the collection of accurate meta data to aid their users in searching for appropriate assets. Any new processes or technologies introduced for reuse libraries must be based on the latest distributive Internet technologies to have any chance of wide spread adoption.

It can be noticed that there is an unfortunate lack of stringent quality control or certification on the assets installed in these libraries. Why is this? Most reuse libraries, both commercial and government, must be self supporting through cost reimbursable fees. Thus the services they provide must be ones that their customers in a capitalistic market place demand. It is not clear why reuse customers are not demanding even minimal assurances about the software they are reusing. One issue could be the lack of legal clarity as to who is responsible and liable for incorrectly performing software. Another is that the labor intensive, manual work currently associated with most reuse libraries' certification processes is too expensive for what is essentially an undocumented monetary return. Quicker, automated certification methods and quantifiable benefits must be discovered and documented if software certification is to have widespread success.

☆U.S. GOVERNMENT PRINTING OFFICE: 1997-509-127-61067

MISSION OF ROME LABORATORY

Mission. The mission of Rome Laboratory is to advance the science and technologies of command, control, communications and intelligence and to transition them into systems to meet customer needs. To achieve this, Rome Lab:

a. Conducts vigorous research, development and test programs in all applicable technologies;

b. Transitions technology to current and future systems to improve operational capability, readiness, and supportability;

c. Provides a full range of technical support to Air Force Material Command product centers and other Air Force organizations;

d. Promotes transfer of technology to the private sector;

e. Maintains leading edge technological expertise in the areas of surveillance, communications, command and control, intelligence, reliability science, electro-magnetic technology, photonics, signal processing, and computational science.

The thrust areas of technical competence include: Surveillance, Communications, Command and Control, Intelligence, Signal Processing, Computer Science and Technology, Electromagnetic Technology, Photonics and Reliability Sciences.